Matti Ala-Hynnilä

Replacing internal communication protocol in UNIC control system

School of Technology and Innovations
Master's thesis in technology
Information- and automation technology

Vaasa 2022

UNIVERSITY OF VAASA
School of Technology and Innovations
Author: Matti Ala-Hynnilä
Title of the Thesis: Replacing internal communication protocol in UNIC control system
Degree: Master of Science
Programme: Information- and automation technology
Supervisor: Jarmo Alander
Instructor: Staffan Tunis
Year: 2022 **Pages:** 86

ABSTRACT:
This thesis examines OPC-UA (Open Platform Communications Unified Architecture) and investigate how it could be used in Wärtsilä for performing internal communication on their UNIC engine control system. Features of OPC-UA are compared to the currently used in-house built protocol to find out if changing the protocol would be feasible.

OPC-UA is a communication specification that standardizes information exchange of industrial automation. This thesis introduces the key concepts of the specification such as information modelling, client-server communication model, publish-subscribe communication model, and the available transportation mappings defining the concrete protocols for transportation. In addition, the current communication implementation of the control system and the services provided for system software components are inspected. After inspections, a general mapping is made between the currently provided services and the OPC-UA features. It is also discussed what transportation protocols shall be chosen for OPC-UA.

The objective of the thesis is to list requirements for performing internal communication by using OPC-UA. Requirements are set for the OPC-UA software development kit features based on the mapped services and protocols. The mapped protocols also introduce requirements for the network stack of the platform software. Based on the feature mappings an architectural proposal for OPC-UA implementation on the control system is presented. It is shown how the different OPC-UA software components could be distributed between the different hardware modules of the system, how the information model and communication interfaces could be initialized in the source code, and how the servers of the different hardware modules could be aggregated into a single server. It is also presented how the information model of the control system could be structured.

A short performance comparison is performed by comparing the data frame structure of the current implementation and the mapped counterpart. Finally, it is concluded that in theory OPC-UA is feasible for performing the internal communication as it provides a lot of options for implementing the tasks of the current service handlers, but in practice the change contains some risks such as immaturity of the technology. Furthermore, the change would require a lot of work, and it could be questioned if the business value of the protocol change is worth the investment.

KEYWORDS: engine control system, distributed system, data exchange, communication protocol, OPC-UA

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen yksikkö
Tekijä: Matti Ala-Hynnilä
Tutkielman nimi: Replacing internal communication protocol in UNIC control system
Tutkinto: Diplomi-insinööri
Oppiaine: Informaatio- ja automaatiotekniikka
Työn valvoja: Jarmo Alander
Työn ohjaaja: Staffan Tunis
Valmistumisvuosi: 2022 **Sivumäärä:** 86

TIIVISTELMÄ:
Tässä opinnäytetyössä tarkastellaan OPC-UA:ta (Open Platform Communications Unified Architecture), ja selvitetään miten OPC-UA-tiedonsiirtoa voitaisiin käyttää Wärstilän UNIC-moottorinohjausjärjestelmän sisäisen tiedonsiirron suorittamiseen. OPC-UA ominaisuuksia verrataan tällä hetkellä käytössä olevaan yrityksen itse valmistamaan protokollaan, jotta saadaan selville, olisiko protokollan vaihtaminen mahdollista.

OPC-UA on tietoliikennespesifikaatio, joka standardoi teollisen automaation tiedonvaihdon. Tässä opinnäytetyössä esitellään spesifikaation keskeisimmät käsitteet kuten tiedon mallintaminen, asiakas-palvelin-viestintämalli, julkaise-tilaa-viestintämalli sekä käytettävissä olevat tiedonsiirtomenetelmät, jotka määrittelevät konkreettiset tiedonsiirtoon käytettävät protokollat. Tarkasteltavana on myös ohjausjärjestelmän nykyinen viestintätoteutus ja sen tarjoamat palvelut järjestelmän eri ohjelmistokomponenteille. Tarkastusten jälkeen tehdään yleinen kartoitus tämänhetkisten palvelujen ja OPC-UA:n ominaisuuksien välille.

Opinnäytetyön tavoitteena on listata vaatimukset sisäisen viestinnän suorittamiselle OPC-UA:n avulla. Käytettävälle OPC-UA-ohjelmistokehityspaketille asetetaan vaatimukset kartoitettujen palvelujen ja protokollien perusteella. OPC-UA:n tarjoamat protokollat asettavat myös vaatimuksia alustaohjelmiston verkkopinolle. Ominaisuuskartoitusten perusteella esitetään myös arkkitehtoninen ehdotelma OPC-UA:n toteuttamiselle ohjausjärjestelmässä. Ehdotelma osoittaa, kuinka eri OPC-UA-ohjelmistokomponentit voitaisiin jakaa järjestelmän eri laitteistomoduulien kesken, miten tietomalli ja tietoliikennerajapinnat voidaan alustaa lähdekoodissa ja kuinka eri moduulien palvelimet voitaisiin yhdistää yhdeksi palvelimeksi. Lisäksi esitetään miten järjestelmän tietomalli voisi rakentua.

Lyhyt teoreettinen suorituskykyvertailu suoritetaan vertaamalla nykyisen toteutuksen datakehysrakennetta ja kartoitettua vastinetta. Lopuksi todetaan, että teoriassa OPC-UA on käyttökelpoinen sisäisen viestinnän suorittamiseen, koska se tarjoaa paljon vaihtoehtoja nykyisten palvelunkäsittelijöiden tehtävien toteuttamiseen. Käytännössä muutokseen sisältyy kuitenkin riskejä, kuten tekniikan tuoreuteen liittyvä epäkypsyys. Muutos vaatisi paljon työtä ja protokollamuutoksen tuottama liikearvo on hieman kyseenalainen verrattuna vaadittuun investointiin.

AVAINSANAT: moottorinohjausjärjestelmä, hajautettu järjestelmä, tiedonsiirto, viestintäprotokolla, OPC-UA

Contents

Figures

Tables

Abbreviations

AMQP	Advance Message Queue Protocol
API	Application Programming Interface
CAL	Communication Abstraction Layer
COM-10	Communication Module
CCM-30	Cylinder Control Module
DC	Direct Current, Data Container
EDL	Enhance Diagnostic Log
FLC	Field Level Communication
FX	Field eXchange
HDA	Historical Data Access
HAL	Hardware Abstraction Layer
HSR	High-availability Seamless Redundancy
ID	Identifier
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IOM-20	Input/Output Module
IT	Information Technology
JSON	JavaScript Object Notation

MAC	Media Access Control
MQTT	Message Queue Telemetry Transport
OPC	Open Platform Communications
OSI	Open Systems Interconnection
PubSub	Publish-Subscribe
QoS	Quality of Service
RMS	Remote Measurement Service
UNIC	Unified Controls
SC	Secure Conversation
SDK	Software Development Kit
SPA	Synchronous Parameter Access
TCP	Transport Control Protocol
TLS	Transport Layer Security
TSN	Time Sensitive Networking
UA	Unified Architecture
UADP	UA Datagram Protocol
UDP	User Datagram Protocol
UML	Unified Modelling Language
UNIC	Unified Controls
URL	Uniform Resource Locator
VLAN	Virtual Local Area Network
WHSR	Wärtsilä HSR
WMAP	Wärtsilä Modular Architecture Platform
XML	eXtensible Markup Language

1 Introduction

The purpose of this thesis is to investigate the possibilities and requirements of replacing WHSR (Wärtsilä HSR) communication protocol with OPC-UA (Open Platform Communications Unified Architecture) in Wärtsilä's UNIC engine control system. The UNIC is a distributed control system, consisting of multiple hardware modules located around an internal combustion engine to control the different parts and processes of the engine. The control system uses High-availability Seamless Redundancy (HSR) network to establish the connection between the different hardware modules. Today, the communication on the HSR network is done with an in-house built WHSR protocol. This thesis will discuss how the current WHSR communication protocol could be changed to use OPC-UA and what are the benefits and disadvantages for doing the change.

1.1 Objectives

The objective of this thesis is to design a concept for using OPC-UA as the internal communication mechanism of the UNIC control system. The required knowledge is achieved by examining the OPC-UA technology and by studying the current WHSR implementation in the UNIC software. The concept will be designed by mapping the different WHSR parts to the OPC-UA counterparts. The goal is to map the WHSR service handlers to use OPC-UA services.

This thesis will describe the working principles of OPC-UA and what services it offers. Furthermore, the thesis will make an analysis of what could be the applicable parts of OPC-UA for the UNIC system and conclude if it would be beneficial to upgrade WHSR by OPC-UA.

1.2 Motivation

OPC-UA is considered one of the key technologies of the fourth industrial revolution known as Industry 4.0, which is heavily based on information and communication

technologies. One of the main concepts of Industry 4.0 are cyber physical systems that according to Cupek, Ziebinski and Drewniak (2017) are systems combining the physical devices and the virtual internet environment. Cubek, Ziebinski and Drewniak point out that the old technologies used in the physical systems are often limited by their connectivity as they are designed mostly to provide data and services in isolated environments. The idea of cyber physical systems is to open the isolated entities and to expose the data and services to be available in a larger network. This allows grouping multiple systems together to form smart objects such as smart factories or smart control systems.

In simple terms, OPC-UA is an industrial data exchange solution that is able to connect different types of machines together to exchange data by defining standardized communication (Hoppe & Stark, 2019). This means that the different virtual systems of the enterprise, such as Enterprise Resource Planner systems, could connect to the OPC-UA server of a floor level physical system and utilize its services. This is illustrated in Figure 1, where field data can be accessed from IT (Information Technology) networks through an OPC-UA gateway.

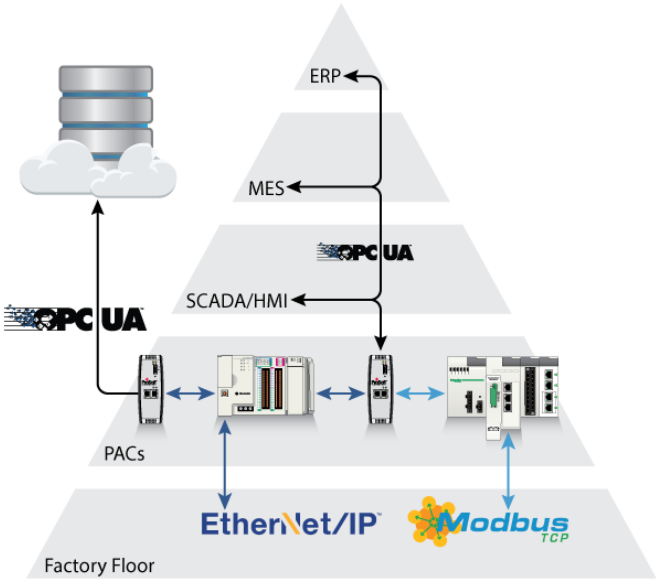

Figure 1. OPC-UA allows standardized communication from floor level gateways to different level IT-systems. (Prosoft, 2022)

Recently, OPC-Foundation has decided to push OPC-UA further towards the factory floor level by working on field level communication (FLC) initiative. The FLC work was announced in 2018, and the objective is to adapt OPC-UA in controller to controller and controller to device communication on the field level (OPC-Foundation, n.d. a). The FLC working groups are working on OPC-UA FX (Field eXchange) specifications, which define the concepts for applying OPC-UA on the field level. These specifications were not yet published during the time of this thesis work but can be expected to be released in the near future, as OPC-Foundation (2021a) has informed that the release candidates of the specifications are in review.

Hoppe (2017) states that OPC-UA is a crucial part of Industry 4.0 as it provides solutions to the challenges of industry 4.0. These challenges include for example standardized data exchange between machines, and data security. Hoppe continues that the Industrie 4.0 platform (a German platform promoting Industry 4.0) requires that a device should support at least the information modelling of OPC-UA to be recognized as an Industry 4.0-ready device.

Wärtsilä has already done some investigations on implementing OPC-UA as a gateway protocol between the engine control system and external systems. This thesis investigates if the UNIC control system should also internally use the OPC-UA protocol as it is designed to be suitable also for embedded environments. When examining the situation from a larger architectural scope, without investigating the OPC-UA properties any further, at least the following results of replacing WHSR with OPC-UA can be considered beneficial:

- changing an in-house protocol to standard one,
- minimizing protocols for the UNIC platform (if OPC-UA is also used for other purposes), and
- the future possibility to connect OPC-UA compatible devices, such as programmable logic controllers, to the control system.

1.3 Thesis structure

This thesis will continue by inspecting how the OPC-UA works and what are the key features that need to be known for starting to design and implement the functionality. Chapter 2 will discuss about the general principles, information modelling concepts, OPC-UA services, technology mapping, and the publish-subscribe model of OPC-UA. In Chapter 3 the UNIC engine control system is presented in more detail and current implementation of WHSR is examined. Chapter 4 will discuss how the current services could be mapped to OPC-UA counterparts and what transport protocol mappings shall be used.

Chapter 5 will continue from mappings defined in Chapter 4 and list the requirements for implementing internal communication of UNIC with OPC-UA. In Chapter 6 an architectural concept is presented how the OPC-UA software could be structured in the control system, and how the information model of an engine could look like. Chapter 7 will briefly compare the frame structures of the different communication models to give an approximation how the OPC-UA upgrade would affect the system performance. Finally, conclusions whether the WHSR replacement with OPC-UA would be realistic are drawn and what directions shall be taken regarding the technology in the future.

To help the reader, this thesis uses a common convention of using *italics* and CamelCase writing style, when writing about OPC-UA terminology and definitions. For example, when talking about *Objects* the term refers to the objects defined in OPC-UA, and object is a general term that can be used for real world objects or object-oriented programming entities. These conventions are also used in the OPC-UA specifications (OPC-Foundation, 2017a).

Another common convention that this thesis uses is the UML (Unified Markup Language) when graphically presenting the OPC-UA information models. The normative UML notations used in this thesis are shown in Figure 2 which is based on the normative definitions shown in annex C of the OPC-UA specification Part 3 (OPC-Foundation, 2022b).

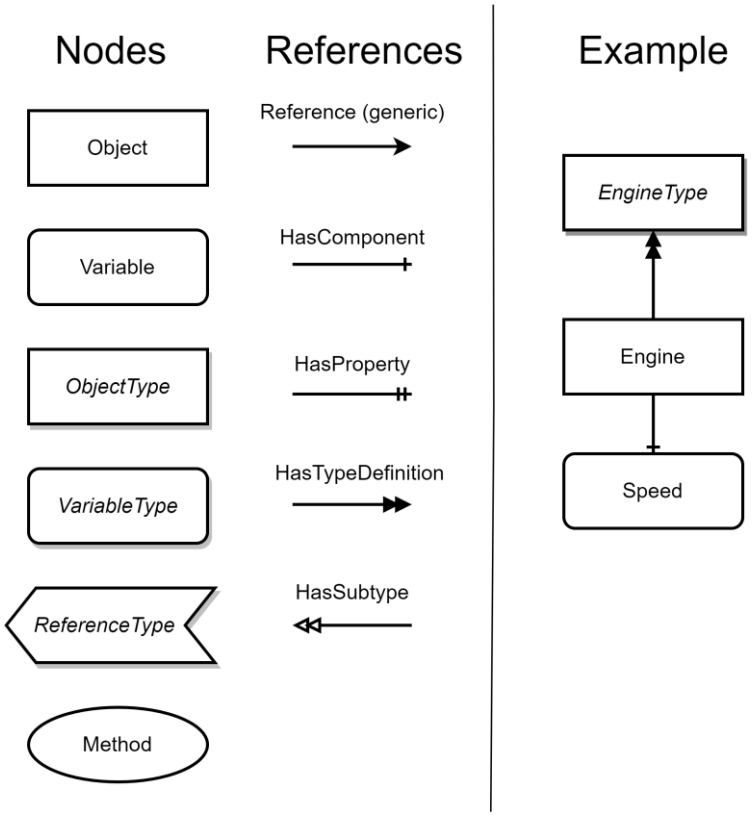

Figure 2. Normative graphical notations used in information model figures.

The OPC-UA information modelling concepts are covered in more detail in Chapter 2.2. A general rule of graphical notations is that *TypeDefinitionNodes* have the same shape as the *Node* they are defining, but have name in *italic*, and have a shadow under the shape.

2 Open Platform Communications - Unified Architecture

OPC-UA is developed and maintained by OPC Foundation which is a consortium creating and maintaining industrial interoperable communication standards. OPC Foundation (2022a) describes OPC-UA as “a platform independent service-oriented architecture that integrates all the functionality of individual OPC Classic specifications into one extensible framework”. In other words, OPC-UA combines all preceding specifications of OPC Classic into one unified specification and generalizes the OPC Classic specifications to achieve platform independence.

The OPC-UA specification is also known as IEC (International Electrotechnical Commission) 62541 standard. OPC-Foundation (2017a) splits the multipart specification into three categories. Parts 1 to 7 and 14 form the core specification determining the base functionality of OPC-UA. Parts 8-11 are the access type specific parts, which focus on the information modelling aspects. Parts 12 and 13 are considered utility specifications, dealing with discovery server functionality and aggregate functions. The specifications are currently evolving, and more parts are to be added such as the field level OPC-UA FX specifications.

2.1 Software layers

OPC-UA software is typically divided into three layers, communication stack, SDK (Software Development Kit), and the client/server application. The software layers are illustrated in Figure 3. On the top is the application layer, which contains the software that exposes and consumes data that is transferred with OPC-UA. Below the application software is the SDK that implements the general OPC-UA functionalities abstracting technical details from the application programmer. Mahnke et al. (2009) also mention that the SDK can also provide non OPC-UA related, common functionality, such as loggers and configuration services.

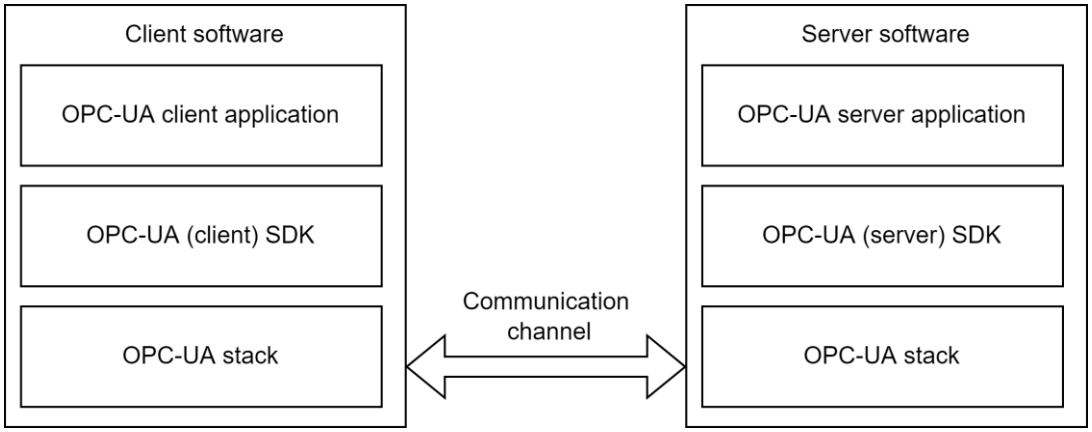

Figure 3. OPC-UA software layers and communication over a channel.

The bottom layer of the OPC-UA application software is the communication stack which is responsible for the communication details. The communication stack layer implements the transport mappings, known also as protocol bindings, defining how the abstract service messages are structured in practice when sent through the communication channel. The stack layers and available technology mappings are inspected in Chapter 2.4. (Unified Automation, n.d.)

The OPC-UA software is not limited to a certain development environment or a programming language. There are multiple SDK implementations (which also include the communication stack software), both commercial and open source. Implementations are available for example with C, C++, Java, JavaScript, C#, and Python (open62541, 2021a).

2.2 Information modelling

OPC-UA follows object-oriented programming principles, where real world entities are modelled with objects. Objects can have various amounts of variable attributes describing them, they can have relations to other objects, and can have methods describing functionalities the object has. The object model of OPC-UA is shown in Figure 4. An object can also have references to other objects or act as an event listener.

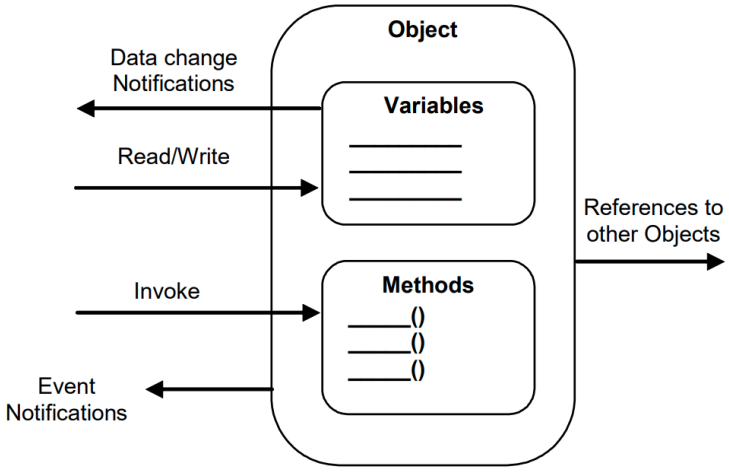

Figure 4. OPC-UA object model used for modelling real world objects. (OPC-Foundation, 2022b)

Mahnke et al. (2009) state that data modelling is the second fundamental concept of OPC-UA beside the data transportation mechanism. According to them, the data modelling defines the base concepts for exposing information in a standardized way. The set of these base building blocks is referred with many names such as *OPC-UA base information model* (Industry40tv, 2020), *MetaModel* (OPC-Foundation, 2022a), or *AddressSpace model* (Mahnke et al., 2009). For consistency, the base model will be referred as *MetaModel* in this thesis. *MetaModel* provides the basic rules for modelling information and the primitive datatypes. On top of the *MetaModel* are the extending information models which define the application and model specific types and constraints for creating the information instances.

2.2.1 Meta model

The *MetaModel* is the basis of every OPC-UA information model. It defines the principal concepts for data modelling such as *NodeClass* definitions and the primitive type definitions. *MetaModel* also provides entry points to the *AddressSpace*, which is the memory area containing the modelled and exposed information. This means that the hierarchical data models are accessed from a certain root entity. (Unified Automation, n.d.)

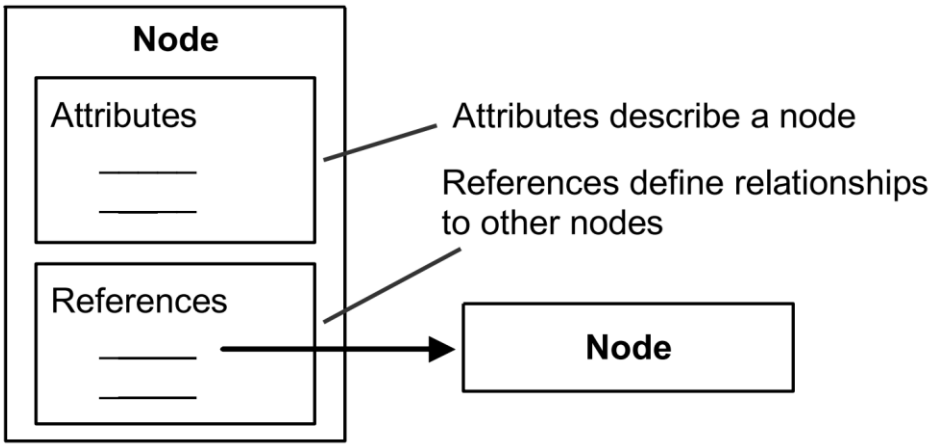

Figure 5. Node model. Node is the basic data structure of OPC-UA. Information structures such as *Objects* and *Variables* are *Nodes.* (OPC-Foundation, 2022b)

The two basic building blocks of OPC-UA data models are *Nodes* and *References* between *Nodes*, which is illustrated in Figure 5. A *Node* is a block of information, containing attributes that are determined by its *NodeClass*. *Nodes* shall not be confused with *Objects*, which are just one type of *Node*. Every *Node* belongs to a certain *NodeClass* such as *Object*, *DataType*, or *Variable*. Industry40tv (2020) says that the *BaseNodeClass* defines some attributes, that are common for all *Nodes*. For example, all the *Nodes* in the *AddressSpace* need to have a unique *NodeId*, a *DisplayName*, a *BrowseName* and they have a *NodeClass* definition. All *Nodes* can also have common optional attributes like *Description* attribute. The *NodeId* datatype contains the information that which namespace the *Node* belongs to. A server manages namespaces with a *NameSpaceArray,* where index 0 contains the *MetaModel* (OPC-Foundation, 2022b).

In addition to the attributes defined by the *BaseNodeClass*, the *Nodes* have also their *NodeClass* specific attributes. The *NodeClasses* and their specific attributes are shown in Figure 6. There are in total 8 different *NodeClasses*: *ObjectType*, *Object*, *VariableType*, *Variable*, *Method*, *View*, *DataType*, and *ReferenceType*. (OPC-Foundation, 2022b)

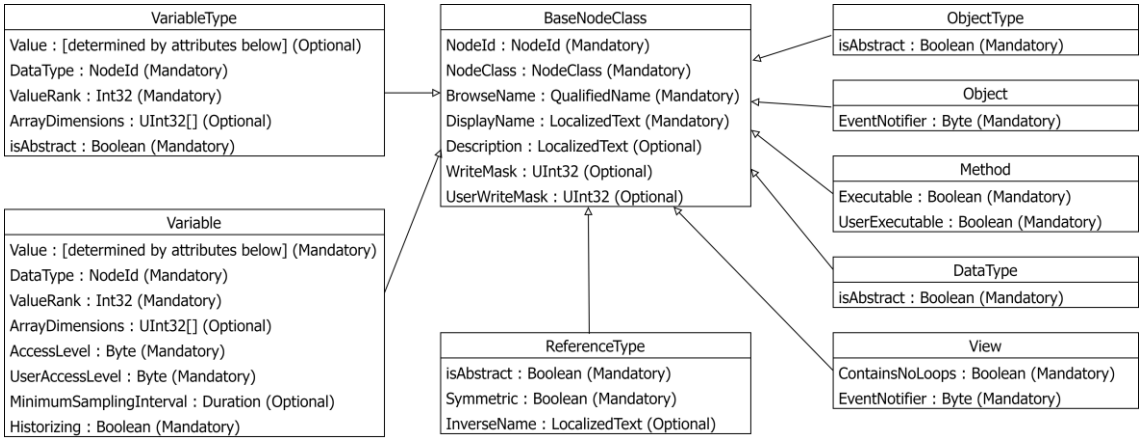

Figure 6. *NodeClasses* and their attributes according to OPC-Foundation (2022b).

Industry40tv (2020) says that the *ObjectType* and *VariableType Nodes* can be considered as classes in object-oriented programming, which are used as templates to create *Object* and *Variable* instance *Nodes*. The *ObjectType* and *VariableType Nodes* form their own type hierarchies, where all defined *ObjectTypes* inherit from the root *BaseObjectType* and *VariableTypes* inherit from *BaseVariableType*. The type definitions can also be declared as abstract (OPC-Foundation, 2022b). Abstract types can be used for inheritance purposes and cannot be instantiated. A simplified diagram of the type definition hierarchies is shown in Figure 7, where the base of the hierarchy is formed with standard definitions of the *MetaModel*, which can be extended to create custom types. In Figure 7 example, some of the standard definitions are extended with arbitrary DC (Direct Current) motor related types.

References connect *Nodes* and exposes a semantic between them. A reference can be used for example to connect a *SuperType* to its *ChildType* with *HasSubType* reference, or an *Object* to its member *Variable* with *HasComponent* reference. Different semantics are defined by different *ReferenceTypes* defined in the *ReferenceType* hierarchy also illustrated in Figure 7. *ReferenceType* can also be abstract. *ReferenceType* has also an attribute named *Symmetric*, which indicates if the semantic of the reference is the same for both directions. (OPC-Foundation, 2022b)

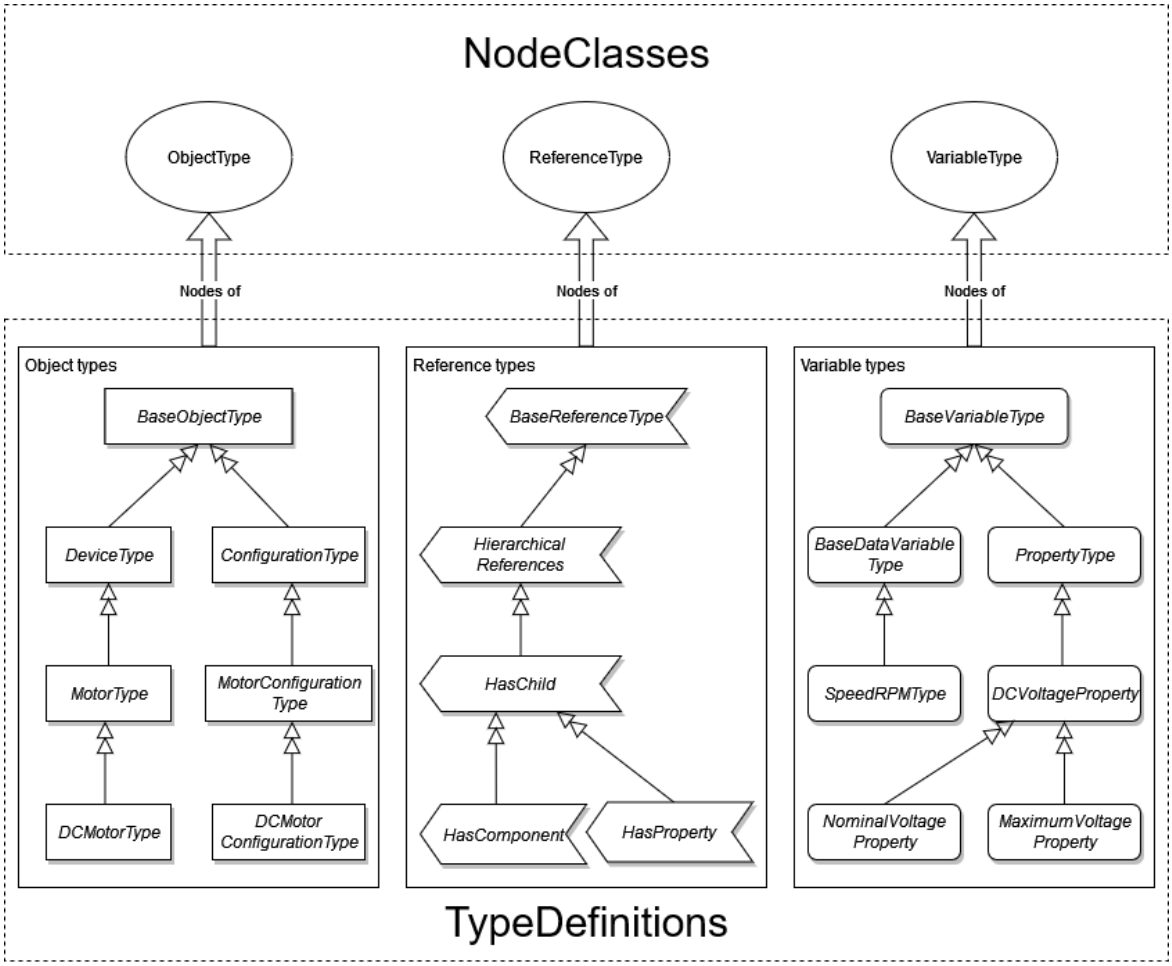

Figure 7. *TypeDefinition* hierarchies inherit from the base types and can be extended with new *TypeDefinitionNodes*.

Mahnke et al. (2009) consider that *Object*, *Variable*, and *Method* are the most important *NodeClasses* of OPC-UA. The concept follows closely object-oriented programming principles. *Objects* are used for structuring the address space, and they consist of *Variables* and *Methods*. *Variables* are *Nodes* for presenting values. In OPC-UA there are two types of *Variables*, *DataVariables*, and *Properties*. A *DataVariable* is used for presenting a changing process value such as speed or temperature measurements. A *Property* is used for presenting a static value, like a constant. A *DataVariable* can have a *Property* (which is a *Variable* connected with a *HasProperty Reference*). For example, a temperature value *Variable* could have a unit *Property*. (OPC-Foundation, 2022b)

Methods are functionalities of an *Object* that the client can call with defined input arguments and expect output arguments to be returned as a result. A motor *Object* can have for example a start and a stop *Methods* for operating the motor. As in object-oriented programming, *Objects* can also reference to other *Objects*, but in OPC-UA *References* also have types describing the relation between the *Objects*. (OPC-Foundation, 2022b)

2.2.2 Modelling example of a motor

Now the arbitrary DC motor related types introduced in Figure 7 can be used to model a simple DC motor. In Figure 8 it is shown how the *DCMotorType Node* could be structured, and how its instance *Object* would be created. As said *ObjectType* has *References* to all its member *Objects* and *Variables*. In this example a *DCMotorType ObjectType* would consist of a speed *Variable*, and a configuration *Object* containing *Properties* of the motor. The *Nodes* that are referenced by a *TypeDefinitionNode*, are special *Object* and *Variable Nodes* called instance declarations (OPC-Foundation 2022b). An instance declaration should also have a reference to a *ModellingRule*, which defines if the member *Object* or *Variable* is mandatory for an instance of the *TypeDefinitionNode* (OPC-Foundation 2022b). When an Instance of the type is created, dynamic versions of the *Objects* and *Variables* are assigned for the instance to hold the instance specific data.

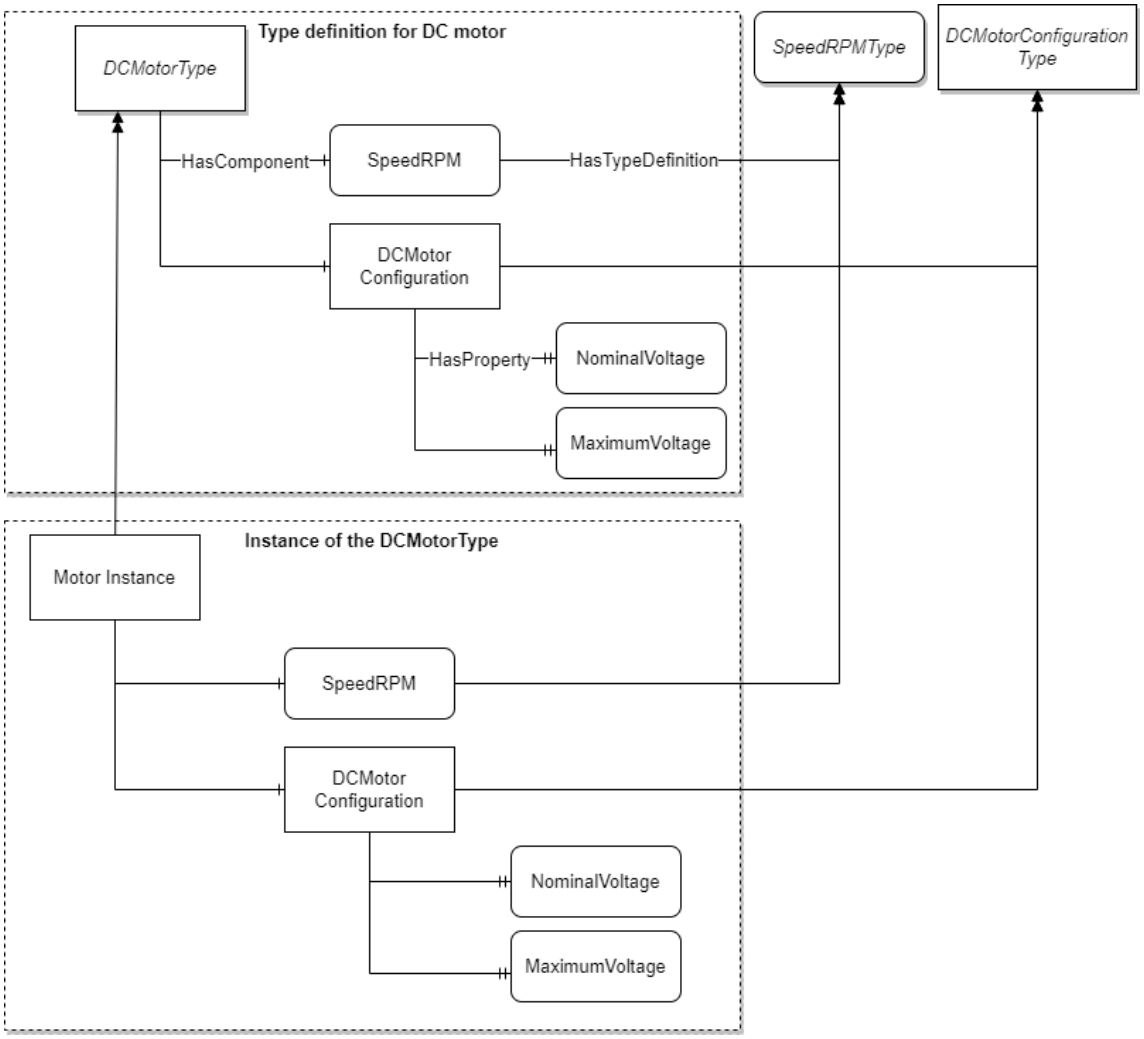

Figure 8. Example of a DCMotorType structure using the types defined in Figure 7. The *ObjectType* is used for creating an *Object* instance.

Nodes of the *DataType NodeClass* are for defining *DataTypes* used in the *DataType* attribute of *Variables*. OPC-UA *MetaModel* defines a variety of built-in data types like different size integers, string, and OPC-UA specific datatypes such as *NodeId*. As with the other *TypeDefinitionNode* hierarchies, the different information models can extend the *DataType* selection with their own *DataTypes*. It is possible in OPC-UA to define enumerations and structures. For new structured types, the encoding needs to be defined, so that the client and server can encode and decode the variables the same way. For base types the encodings are obviously already defined in the OPC-UA specification. (OPC-Foundation, 2022b)

The last *NodeClass* to discuss is the *View NodeClass*. Mahnke et al. (2009) consider *Views* with their two main purposes. First, *Views* are used as entry points to the AddressSpace. This means that a *View* can have references pointing directly to the *Nodes* of interest so that the client does not need to browse through the whole address space from the root to get to the devices of interest. Second, *Views* are used to reduce the number of visible *Nodes* and *References* for the viewer. For example, a maintenance *View* could be created to filter field devices to show only their maintenance data, hiding extra information such as process values, for which a maintenance person would have no interest. The *View Node* could have, for example in the maintenance view case, references to all the devices that the current viewer is responsible for maintaining.

One of the key features of OPC-UA are *EventsNotifiers*, which are similar to event listeners used in most programming languages. An *EventNotifier* is used to follow certain events and their occurrence. When an *Event* occurs the *EventNotifier* can send the *Event* information with a *Notification*. To receive *EventNotifications*, a client subscribes for the *EventNotifier* with subscription services discussed in Chapter 2.3.3 or by using the publish-subscribe model introduced in Chapter 2.5. The type of an *EventNotifier* is defined with an *EventType*. There is no separate *NodeClass* for *EventNotifiers* or *EventTypes. EventsTypes* are rather *Nodes* of *the ObjectType NodeClass* and the *EventNotifiers* are *Object Nodes*. To define *EventNotifiers*, *Object* and *ObjectType Nodes* have a special *EventNotifier* attribute. The *BaseEventType*, which is the root of the *EventType* hierarchy, inherits directly from the *BaseObjectType*. (OPC-Foundation, 2022b)

2.2.3 Extending information models

Whereas the *MetaModel* is the base of every OPC-UA application, the extending information models are used for application specific modelling (Unified Automation, n.d.). As an example of extending the *MetaModel*, in Figure 7 custom motor *ObjectTypes* were inherited from the base types. Instead of just inheriting all *ObjectTypes* from the *BaseObjectType* defined by the *MetaModel*, it is more powerful to create hierarchies by inheriting subtypes and using abstract types. An information model could define an

abstract motor type, which is used as a parent *ObjectType* for all the different types of motors, and it would include properties that are common to all motors. The information models are designed to be extensible, and multiple information models can extend each other to create the modelling rules.

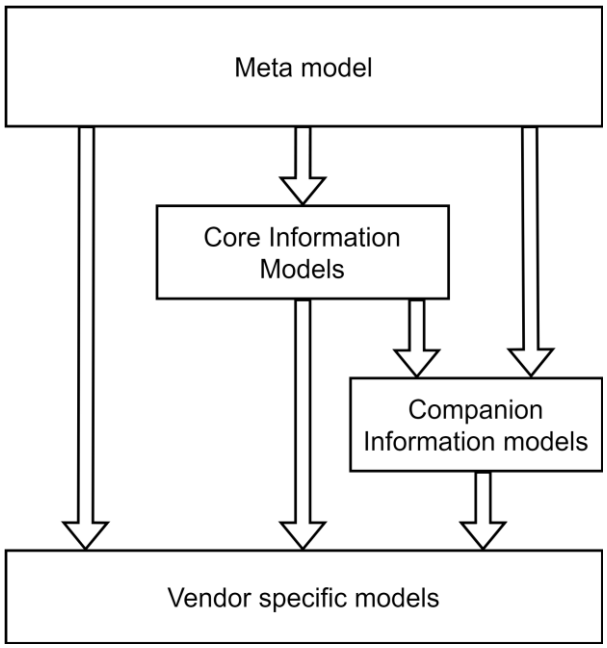

Figure 9. Information models extend the MetaModel and other more generic models.

OPC-Foundation (2022a) considers 3 different layers of information models extending the *MetaModel*. These model layers are visualized in Figure 9, where on the top the layers offer more generic modelling, and they are extended with the arrows downwards where layers give more precise definitions. First, there are the core information models produced by OPC-Foundation. Core information models are very general information models that have wide area of use cases. Industry40tv (2020) describes that the 4 main core information models are Data Access, Historical Access, Alarms and Conditions, and Programs, which are adaptations of the Classic OPC. These information models are defined in Parts 8-11 of the specification.

Second are the companion information models which define industry specific modelling standards (OPC-Foundation, n.d. b). For example, programmable logic controller

manufacturers could agree upon certain modelling rules to make standard compatible devices to increase interoperability between devices from different vendors. Last are the vendor specific information models, which allow vendors to create custom information modelling rules for their own needs. It shall be noted that the vendor and companion specific models could be extended right from the *MetaModel*, or by extending layer by layer to produce interoperability on different levels. (Unified Automation, n.d.)

2.3 Services

OPC-UA uses service-oriented client-server architecture, where client components request services from server components. A server has a defined set of services, which the client can call, also known as making a request, for which the server answers with the service specific response. The services are grouped to different service sets according to their purpose. Industry40tv (2021) says that a service consists of a request and a response and that all services use common request and response headers. A simple example of a service would be the read service, where client defines the information it wants to read in the request, and the server responses with the requested data.

The OPC-UA specification defines in total 39 abstract services. The service definitions are abstract, because they shall be independent from the various transport protocols, encoding schemes, and programming environments. The mapping of abstract services to concrete transportation mechanisms is defined in Part 6 of the OPC-UA specification and is discussed in Chapter 2.4 of this thesis. (OPC-Foundation, 2021b)

According to Industry40tv (2021), one important aspect of OPC-UA services is that they are stateful, meaning that communication context needs to be established on different levels. This is the reason why most of the services are used for managing the communication infrastructure. The communication context layers of the client-server communication model are visualized in Figure 10.

In short, the lowest communication context is established on the transportation channel, where on top OPC-UA builds a *SecureChannel* context defining the security aspects of the connection. A *SecureChannel* is created and maintained with *SecureChannel Service Set*, and it is handled by the OPC-UA communication stack (Industry40tv, 2021). On top of *SecureChannel* is the *Session* layer, which requires the application layer to utilize the *Session Service Set* to open and maintain a communication session. When a *Session* is opened on a *SecureChannel* the further service requests are bound to the formed *Session*. A *Session* can contain *Subscriptions*, managed with the *Subscription Service Set*, and a *Subscription* consist of *MonitoredItems*, which are managed with the *Monitored Item Service Set*. The concept of *Subscriptions* is further discussed in Chapter 2.3.5.

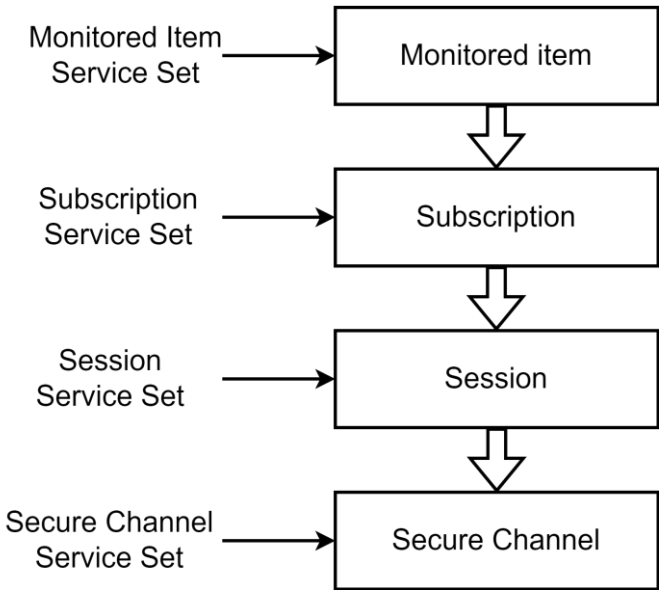

Figure 10. Communication context layers of OPC-UA described by Mahnke et al. (2009)

The next subchapters will introduce the OPC-UA client-server model services by grouping the *Service Sets* by their use cases. These service groups are the address space browsing services, read and write services, subscription services, method calling services, historical read and write services, query services, and services for modifying the address space. The discovery services and services regarding connection management are not discussed, as they are usually handled by the SDK, and are hidden behind an abstract API (Application Programming Interface).

To achieve efficient data transfer, the OPC-UA services allow bulk operations (Industry40tv, 2021). It means that most of the services, especially the ones that are used for information access, can process multiple items in a single request. For example, with the *Read service* request the client can specify multiple *Nodes*, which it wants to read, and all the read results are returned in a single response.

2.3.1 Finding information from servers

Services from the *View Service Set* are used for browsing the servers *AddressSpace*, so that a client can find *Nodes* that it is interested on. *Browse service* is used to navigate in the *AddressSpace*. The *Browse service* request contains a set of starting *Nodes* and a set of filters such as the *ReferenceTypes* the browsing should follow and into what direction of the hierarchy. Also, the request can contain a *View* parameter to restrict the visible *Nodes* and a maximum limit to limit the number of *Nodes* to return. In response the server returns the set of *References* associated with the requested *Nodes*, and the information of the *Nodes* they are connected to. (OPC-Foundation, 2021b)

The idea of browsing the *AddressSpace* can be visualized by thinking the *Nodes* as a network connected with *References*. This is illustrated in Figure 11. The *Browse service* can be used to hop from *Node* to *Node* by following a *Reference*. An application can create apply *Views* to restrict the amount of browsable *Nodes*.

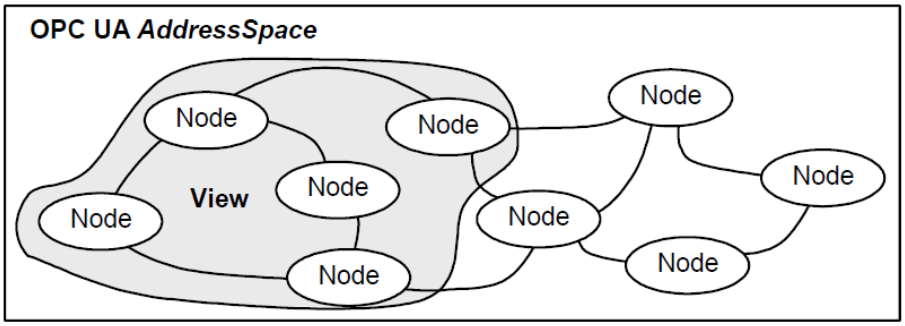

Figure 11. OPC-UA address space visualized as a network of unique *Nodes* identified by their *NodeId*. (OPC-Foundation, 2017a)

In case the *Browse service* would return more *Nodes* that is limited by the request or by the server, the response tells the client that there are more *Nodes* available, which the client then reacts with a *BrowseNext* service request. In such case, the *BrowseNext* service request is mandatory even if the client is not interested in retrieving more *Nodes*. The *BrowseNext* request contains a particular flag indicating that client is not interested on receiving anymore *References* to *Nodes*. (OPC-Foundation, 2021b)

RegisterNodes and *UnregisterNodes services* are used for registering *Nodes*. A *Node* can be registered for optimization when a client wants to access the *Node* more frequently. Registering *Nodes* optimizes the access in two ways. Most importantly, the server can optimize its access to the source data. OPC-Foundation (2021b) also recommends creating a numeric alias *NodeId* (handle) that could be used instead of the long *NodeId* datatype.

2.3.2 Read and write

Read and *Write services* are basic services used by the client to read and write *Node* attributes in the server *AddressSpace*. The *Read service* is viable when information needs to be read irregularly, since regular polling of value changes would be a waste of resources. For values that need to be constantly updated, like changing process values, the OPC-UA uses the concept of a *Subscription*, which is discussed in the next Chapter. (OPC-Foundation, 2021b)

2.3.3 Subscriptions

A *Subscription* contains *MonitoredItems*, that the subscribing client is interested in. The three different *MonitoredItem* types which the client can add to a *Subscription* are *Variable* values, *Events*, and *AggregatedVariable* values. Now the client does not need to poll the *Variable* values with the *Read service*, but it can select the *MonitoredItems* into a *Subscription* and the server will send *Notifications* when the values change (Zunino et. al. 2021). A client can create multiple *Subscriptions* in one *Session* context and a

Subscription can contain multiple *MonitoredItems*. *Subscription* structure is presented in Figure 12 visualizing the different parameters of a *Subscription* and its MonitoredItems.

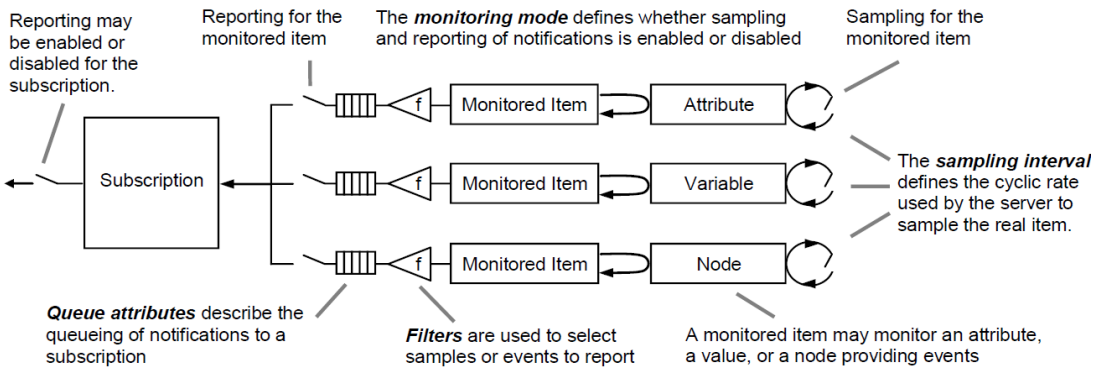

Figure 12. Subscription structure with monitored items. (OPC-Foundation, 2021b)

Subscriptions have two important parameters. First, the *RequestedPublishingInterval* determines the time interval for the server for combining buffered *Notifications* produced by *MonitoredItems* into a *NotificationMessage* that is sent to the client (Zunino et. al. 2021). Second, the *PublishingEnabled* determines if the *Subscription* shall send *NotificationMessages* to the client or not.

A *Subscription* is created with the *CreateSubscription service*. The sending of *NotificationMessages* can be toggled on/off with the *SetPublishingMode service*. *DeleteSubscription service* is used for deleting subscriptions. Last service for managing *Subscriptions* is the *TransferSubscription service*, which can be used to transfer *Subscriptions* to a different *Session* context. For example, a redundant client can call this service from the new *Session* to retrieve the *Subscription* of a failed client. When a *Subscription* is created, the client can add *MonitoredItems* to it with the *CreateMonitoredItems service*. In addition to the *NodeId* and *AttributeId* for determining the item to monitor, there are two important fields. The *MonitoringMode* parameter determines if the item is sampled and if the occurring *Notifications* shall be included in the published *NotificationMessages*. The *MonitoringParameters* determine the sampling interval, filters, queue size, and discard policy, visualized in Figure 12. (OPC-Foundation, 2021b)

Created *MonitoredItems* in can be modified in with the *ModifyMonitoredItems service* and deleted with the *DeleteMonitoredItems service*. Similarly, as in the *Subscription service set*, the *SetMonitoringMode* service is used for toggling the sending of notifications and sampling on/off. It is also possible to set a *MonitoredItem* to send notifications with a trigger from separated *MonitoredItem*. The trigger is set by using the *SetTriggering* service, which the client can use for adding and removing links between the triggering *MonitoredItem* and the *MonitoredItem* producing notifications. (OPC-Foundation, 2021b)

A simplified sequence diagram is shown in Figure 13 illustrating the sequence of service calls for creating a subscription and publishing information. When a client has created a *Subscription* and added the *MonitoredItems*, *Publish* and *Republish services* are used for delivering the notification messages. *Publish service* request acts as a life ping that the client is requesting *NotificationMessages* from the *Subscription*. The request contains a sequence number of the latest *NotificationMessage* acknowledged, which the server can use to check if there are any lost messages or if there are newer messages in the queue ready for sending. (OPC-Foundation, 2021b)

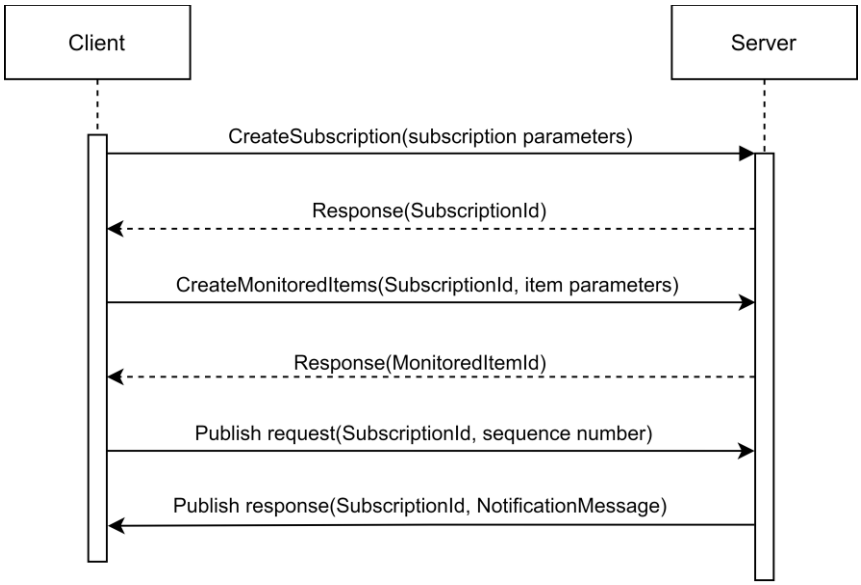

Figure 13. Sequence diagram showing process of creating a subscription.

Mahnke et al. (2009) mention that the *Publish service* is a special type of service regarding the fact that it is the only service where the server can ignore a request without any processing. It is expected that the server publishes *NotificationMessages* within the intervals determined by Subscriptions, not as an immediate response. If there are no *Notifications* to be sent in a scheduled *NotificationMessage*, the server sends an empty *NotificationMessage* which also acts as a life ping towards the client for maintaining the connection.

2.3.4 Calling methods

Call service is used for calling a *Method* of an *Object*. Mahnke et al. (2009) remind that the client needs to have built-in knowledge or gain the information by browsing the address space, to know the *Method* arguments. In the *Call service* request the client defines the *NodeIds* of the *Object* and *its Method* to be called, and the input arguments for the call. The service response contains results such as status code for the call and the output arguments. Again, bulk operations are supported, so multiple *Methods* from multiple *Objects* can be called in a single request. Figure 14 illustrates how the *Method* gets executed on the server in an assigned callback function. (OPC-Foundation, 2021b)

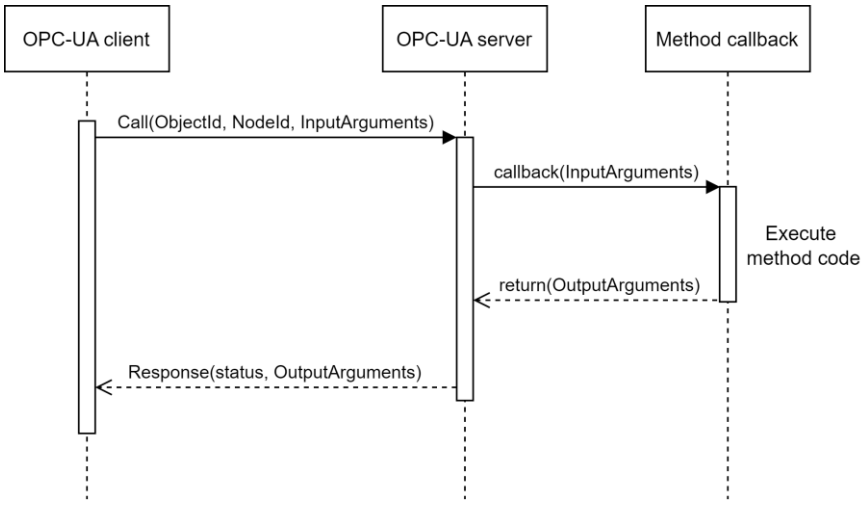

Figure 14. Flow of the *Call service.* A method is assigned with a callback function, that is executed when a call is made.

2.3.5 Reading historical data and events

Historical data can be read with the *HistoryRead service* and the *HistoryUpdate service* is used to insert, modify, or delete historical data. *HistoryRead* request uses an extensible parameter for defining if the request is for reading raw data, modified data, processed data, if the request defines a set of timestamps to read, or if the read is concerning history of events (Mahnke et al. 2009). The *HistoryRead* response uses either *HistoryData* or *HistoryEvent* extensible parameter for sending the result according if the read was for data or event history (OPC-Foundation, 2021b).

The difference to normal read and write services is that the historical data of *Variables* and *Events* is not stored in the server *AddressSpace* and that the messages contain history specific parameters. A separate storage mechanism such as database is required that the server accesses when receiving requests. For this purpose, it is common to use separate "historian" software that collects and manages the historical data. Data flow of historical access is shown in Figure 14, where Historical Data Access (HDA) server accesses Proficy historian, a historical access solution of GE Digital (2022).

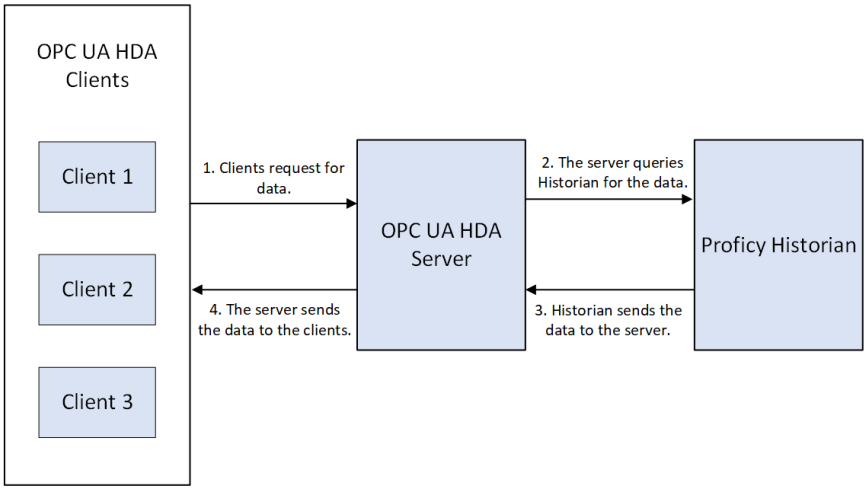

Figure 15. OPC-UA server with historical data access capabilities fetching the requested data from a historian. (GE Digital, 2022)

2.3.6 Query

In addition to the browsing services discussed in Chapter 2.3.3, OPC-UA has also *Query services* for finding information from a complex *AddressSpace* that could have up to millions of *Nodes*. In a such complex *AddressSpace* the concept of browsing, searching from *Node* to *Node*, is not sufficient. The *Query service* principles are similar to database query languages, where a subset of entries is retrieved by specifying some filtering criteria. The OPC-UA queries are used to find instances of a *TypeDefinitionNode* (OPC-Foundation, 2021b). A query could for example be used to get all instances of a certain motor *ObjectType*. Filtering criteria could be used to for example get only the motors that are currently running or require maintenance.

There are two query services, *QueryFirst* and *QueryNext*. Like in browsing, the *QueryFirst* is the service used to do the actual querying and *QueryNext* is used to continue retrieving more results of the query if the number of results exceeds the limits. The data can be filtered by passing a *View* and attribute content filters in the *QueryFirst* service request. (OPC-Foundation, 2021b)

2.3.7 Modifying the information model

The server *AddressSpace* can be dynamically modified with the *NodeManagement service set*. A new *Node* can be created with *AddNode service* and *Nodes* can be deleted *DeleteNodes* service. A *Reference* between two existing *Nodes* can be created with the *AddReferences service*. *DeleteReferences service* can be used to delete existing references. (OPC-Foundation, 2021b)

2.4 Transportation mapping

As the OPC-UA services are abstract definitions, they can be implemented with multiple different encoding and transportation technologies that define the required implementation details. The abstract definitions also give room for future technologies, which can be added to the OPC-UA specification later (OPC-Foundation, 2017a). Part 6 of the OPC-

UA specification defines the available protocols, known as *Mappings*, wherefrom at least one technology shall be chosen for each layer of the communication to be OPC-UA compliant. For OPC-UA applications the information of supported technologies is handled with *Profiles* which are standard definitions for expressing feature compatibility.

The Mappings are divided to three layers, data encoding, security protocols, and transport protocols (OPC-Foundation, 2022c). These three layers form the communication stack of an OPC-UA application. The current situation of the available *Mappings* is illustrated in Figure 16. The next subchapters will discuss briefly about the currently available technologies on each layer of the stack.

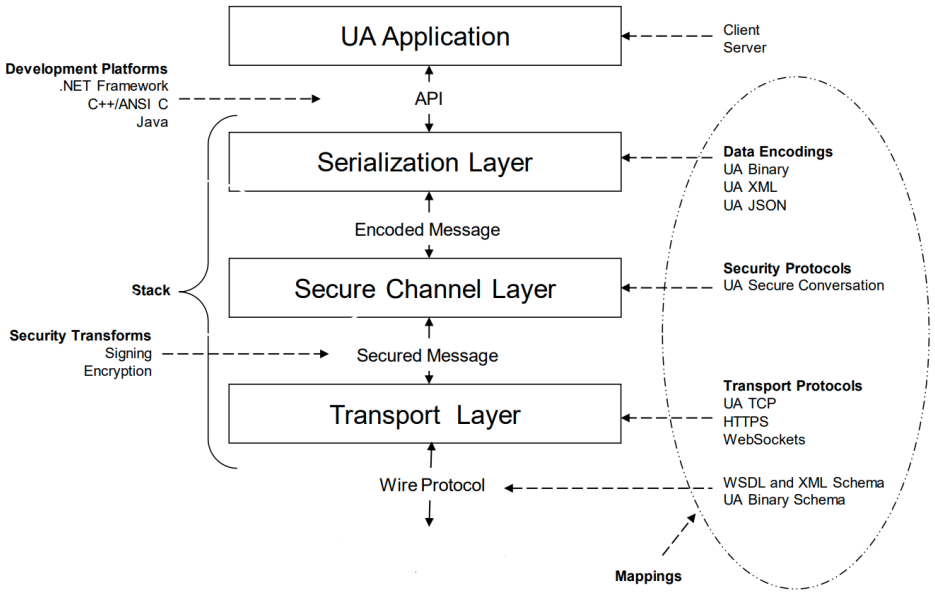

Figure 16. OPC-UA stack layers and available mappings in the specification (OPC-Foundation, 2022c).

2.4.1 Data encoding

OPC-UA standard has three available encoding schemes, *UA Binary*, *UA XML*, and *UA JSON* (Unified Automation, n.d.). *UA Binary* is according to OPC-Foundation (2022c) an encoding mechanism designed to achieve high performance. As the name suggests, *OPC-UA Binary* is used to encode data in binary form, meaning that information is

straight serialized into a binary stream with certain rules and binary representations of primitive data types.

UA XML and *UA JSON* use the more universal XML (eXtensible Markup Language) and JSON (JavaScript Object Notation) encoding schemes. The actual encoding is done according to the encoding standard, and the UA prefix references how the OPC-UA specific data types such as *NodeId* shall be presented with the standard encoding (OPC-Foundation, 2022c). The advantage of XML and JSON is that they are already widely supported, but they also cause performance overhead when compared to the binary encoding.

2.4.2 Security layer

UA-SecureConversation known as *UA-SC* is currently the only security protocol in the OPC-UA standard. According to Mahnke et al. (2009) UA-SC is not a completely new protocol defined by OPC-Foundation, but rather a combination of techniques from other protocols such as TLS (Transport Layer Security). Security aspects of OPC-UA are defined in Part 2 of the specification and are not presented in this thesis.

In short, *UA-SecureConversation* appends 4 information blocks on the encoded message payload as shown in Figure 17. First block of the chunk is the message header defining the message type. Second block is the security header, which contains information of the used security algorithms, a certificate for verifying the message signature, and the identification of the certificate used for encrypting the message. Third block of the message chunk is the sequence header, which identifies the chunk numbers when a message does not fit into a single TCP (Transmission Control Protocol) packet. The last block is the security footer placed after the payload that consists of padding and the message signature. (OPC-Foundation, 2022c)

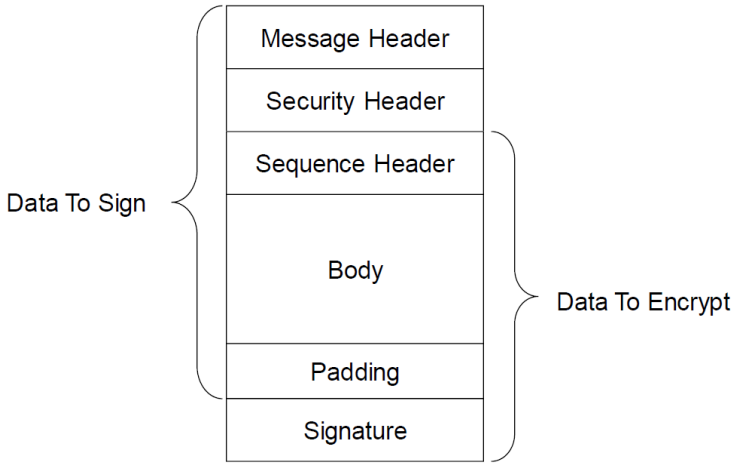

Figure 17. Structure of an UA-SC message chunk. (OPC-Foundation, 2022c)

2.4.3 Transport protocols

OPC-UA offers three possible transport protocol mappings for the client-server communication. The options are *UA TCP*, *UA HTTPS* and *WebSocket*. *UA TCP* is a small protocol on top of regular TCP/IP protocol (Transmission Control Protocol/Internet Protocol). In *UA TCP* protocol data is sent in packets consisting of the payload and the message header. The header describes the message type and length, and the payload is either TCP related handshake message or an encoded and secured service message. The message header is the same as with UA-SC shown in Figure 17. Rinaldi (2019) lists that the responsibilities of UA TCP are opening the communication link between the client and server, negotiating message and buffer sizes, keeping the message chunks in order, and transporting the secured messages received from the security layer between the client and server. (OPC-Foundation, 2022c)

The main use case for *UA HTTPS* and *WebSocket* protocols is in the web environments. The greatest advantage of these protocols is that they are able to pass through firewalls and therefore communicate over different networks. Having the internet capability, these protocols require more software layers on top of regular TCP/IP.

2.5 Publish-Subscribe

In addition to the previously introduced service-oriented client-server model, OPC-UA has a message based publish-subscribe model known as *PubSub*. It can be used when there are multiple network nodes interested on same information. *PubSub* is defined in Part 14 of the specification and shall not be confused with the subscription services of the client-server model. According to Aro (2021) the main disadvantages of the client-server model are scalability and network sensitivity. The client-server model is not scalable, because each client-server connection requires resources to establish and manage the connection on the different communication context layers. What *PubSub* communication allows is publishing data without building any communication context.

The *PubSub* model consists of *Publishers* and *Subscribers*. The principle is that instead of forming one to one connection, information is transferred to clients in a multicasting manner. Aro (2021) describes that *Publishers* define *DataSets*, which can contain *Variables* and *EventNotifiers*. According to OPC-Foundation (2022d) a *DataSet* can be considered as node-value pair list presenting an *Event* or *Variable* values. The list of *Variables* and *EventNotifiers* defining the contents of a *DataSet* is called *PublishedDataSets* for a *Publisher* and *SubscribedDataSets* for *Subscribers*.

Structure of a Publisher is shown in Figure 18. First publisher creates a *PublishedDataSet*, listing what is being published in a *DataSet*. A *DataSet* contains the actual published information written by a *DataCollector* that fetches the information from the *AddressSpace*. A *DataSetWriter* encodes the information of a *DataSet* into a *DataSetMessage*. These *DataSetMessages* are then sent over the network in *NetworkMessages*. In addition to the *Variable* and *Event* information, the *PublishedDataSets* contain also *DataSetMetaData* which describes the *DataSet*. The *DataSetMetaData* is not sent in the published *NetworkMessages* but can be read from the address space using the client-server model services. (OPC-Foundation, 2022d)

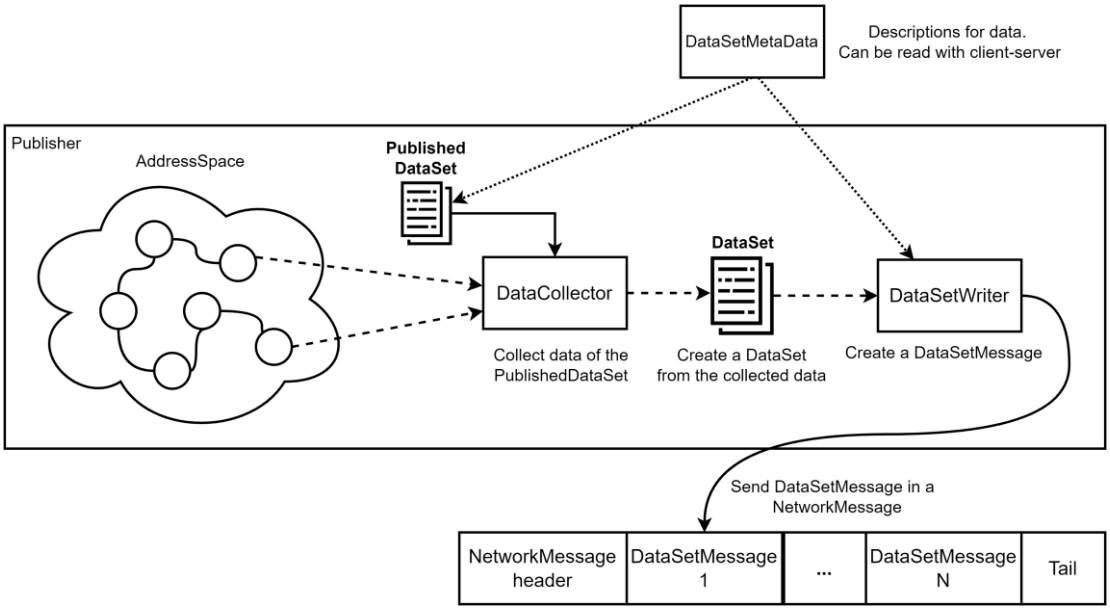

Figure 18. OPC-UA Publisher structure. *PublishedDataSet* contains information what needs to be published, and the *DataCollector* creates *DataSet* accordingly. *DatasetWriter* creates *DataSetMessage* from the *DataSet* and sends it in a *NetworkMessage* to the subscribers.

On *Subscriber* side the information flow is similar to the *Publisher* presented in Figure 18 but mirrored. An additional step for the *Subscriber* is to filter the *DataSetMessages* of a *NetworkMessage* according what it has subscribed for. The *DataSetMessages* of interest are then parsed by a *DataSetReader* to receive the *DataSet* information. Last, the *AddressSpace* of the *Subscriber* is updated by dispatching the data of the *DataSet* according to the *SubscribedDataSet* information. (OPC-Foundation, 2022d)

2.5.1 Network models

Aro (2021) says that OPC-UA defines two *PubSub* network models. In the local network model *Subscribers* and *Publishers* are connected to a local network, where UDP (User Datagram Protocol) broadcast or plain Ethernet is used for transportation. The other network model is the message queue broker in which the network also consists of a broker, a middleman which purpose is to forward published *DataSetMessages* to interested *Subscribers*. Message queue broker can be implemented either with MQTT (Message Queue Telemetry Transport) or AMQP (Advanced Message Queue Protocol). The available

encodings for *PubSub* are binary UADP (Unified Architecture Datagram Protocol) and JSON. (OPC-Foundation, 2022d)

2.5.2 Time sensitive networking

A recent topic in the OPC-UA community is the OPC-UA over TSN, known as Time Sensitive Networking. Time sensitivity is synonym for real-time determinism, the transported messages should have bounded latency with low variation on the travel time known as jitter (Joung & Kwon, 2021). TSN is a set of protocols on the layer 2 of the OSI (Open System Interconnection) model, which purpose is to make standard Ethernet deterministic (Profinet, n.d.). The use of TSN allows taking OPC-UA communication also to the field level devices and processes which are time critical.

In terms of OPC-UA, the TSN is targeted particularly for the *PubSub* Ethernet mapping, although the Ethernet mapping can also be used for regular non-deterministic Ethernet. The *PubSub* specification defines mechanisms for demanding so-called Quality of Service (QoS) for *NetworkMessages* (OPC-Foundation, 2022d). This means, how the network needs to be configured to achieve timing requirements of an application.

The QoS types are defined in the *PubSub* specification in a way that abstracts the network specific capabilities. Currently the specification defines only priority based QoS types. The concept is visualized in Figure 19, where Publisher defines *PriorityLabels* for *NetworkMessages*. These *PriorityLabel* priorities are abstract and need to be mapped to concrete network specific priorities. (OPC-Foundation, 2022d)

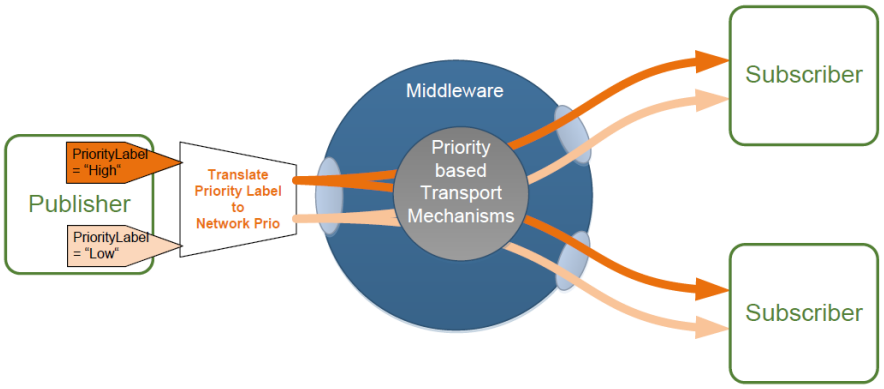

Figure 19. Priority based scheduling with *PubSub*. Higher priority messages are sent before lower priority messages. (OPC-Foundation 2022d)

The network specific parts are modelled with a base network model introduced in the specification Part 22. When the network type and its components are inherited from the base model, the abstract QoS definitions can then be mapped to the network specific QoS functions to configure the network. Currently, the specification focuses on priority based QoS, where priority labels defined on application level are mapped to network specific priorities. The QoS functionalities that utilize TSN, such as timing deadlines are still under development. In general, it seems that the whole QoS mechanism is still in its concept phase, with no real use-case implementations. (OPC-Foundation 2022e)

One reason why TSN capabilities are not yet utilized in OPC-UA is that the listing of TSN protocols to be used with OPC-UA not yet defined. TSN is a wide set of standards defining things like time synchronization mechanism, traffic scheduling algorithms, frame preemption mechanisms, etc. Instead of implementing all standards defined under TSN, the TSN task group (n.d.) defines profiles that pick certain set of the standards for a certain use case. One of these profiles is the IEC/IEEE (Institute of Electrical and Electronics Engineers) 60802, TSN profile for industrial automation. OPC-Foundation (2021c) is looking forward for this standard to reference it for future QoS mappings, and in the upcoming FLC-specification. The problem is that the IEC/IEEE 60802 has not yet been published.

3 Internal communication in UNIC

UNIC, known as Unified Controls, is Wärtsilä's engine control system for their internal combustion engines. UNIC2 is the second generation UNIC control system which is currently the main focus of development in the control system development department. This thesis considers the internal communication of the second generation UNIC system and when used in this thesis, the name UNIC explicitly refers to the UNIC2, as the given information does not apply to the first generation UNIC system.

The UNIC2 engine control system is built with COM-10 communication modules, CCM-30 cylinder control modules, and IOM-20 input/output modules. The UNIC system is modular, so that it can be configured for many of the engine types produced by Wärtsilä regardless of the engine fuel type or the cylinder configuration. Different types of engines can have different number of hardware modules that are running engine configuration specific software. As said, the control system is a distributed system where each module runs its designated control software. Modules share process information such as measurement values in an internal Ethernet network.

The software running on the hardware modules can be divided roughly into two main components. The lower layer software component is the Wärtsilä Modular Architecture Platform known as WMAP, which creates a platform for running the engine control applications considered as the upper layer software. In other words, the WMAP software does not control the engine, but provides the infrastructure such as information management, I/O services, communication handling, etc. for abstracting the low-level technical details from the engine control application software. The topic of this thesis concerns the WMAP platform software, where the internal and external communication mechanisms are implemented.

3.1 Network topology

The modules of UNIC system are connected together with 100Base-TX Ethernet to form a HSR ring network. The HSR protocol is standardized by IEC 62439-3. In HSR network the messages are sent in both direction of the ring in Ethernet frames. Each of the nodes receiving the message will check if the message is meant for them, and either send the message to upper software layers for processing the message or pass the message forward in the ring. Also, if a network node receives an Ethernet frame it has already received from the other side of the ring, it can just discard it, as the frame has already passed through all other nodes of the network. The redundancy aspect of the HSR is obvious. If a single connection in the ring breaks, the message will still be received from other side of the ring. The redundancy of HSR is also invisible to the upper layers. WHSR knows only if a module is reachable or not but has no knowledge about the network topology. The HSR ring formed with the UNIC modules is illustrated in Figure 20. (Wärtsilä, 2019a)

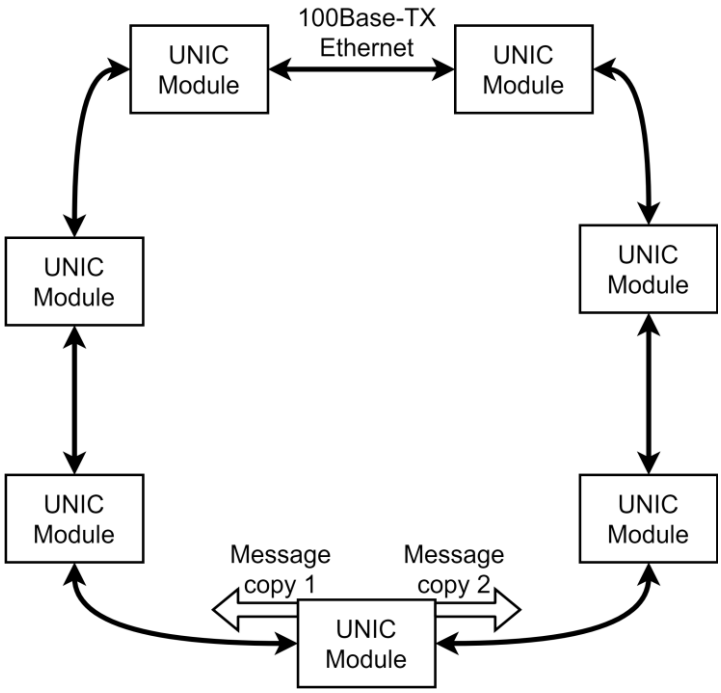

Figure 20. UNIC control system modules form a HSR ring with 100Base-TX connections where messages are sent in both directions.

3.2 Protocol stack

The WHSR protocol consist of two main layers, communication layer, and the so-called middleware layer. The communication layer corresponds to the network layer of the OSI model, and the middleware layer corresponds to the transport layer of the OSI model (Wärtsilä, 2019b). Therefore, when comparing to OPC-UA the WHSR layers would correspond to the transportation layer of the communication stack. The WHSR communication layers are presented in Figure 21. On the top of the Figure are the service handlers utilizing the WHSR stack to handle the application specific communication for the different WMAP software components and abstracting the communication details. A service handler uses the WHSR stack API to register itself to the WHSR to receive certain type payloads and to send data. The service handlers are usually executed in the context of the system software scheduler as a task. (Wärtsilä, 2019b)

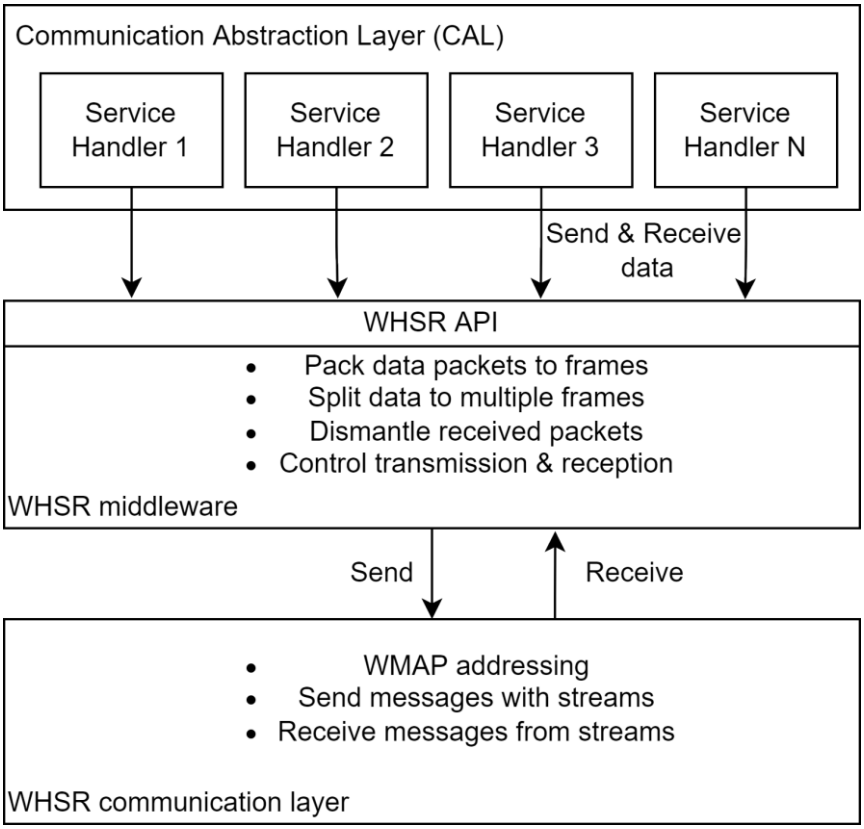

Figure 21. WHSR communication stack layers.

When a service handler makes calls via the WHSR API, the calls are processed by the WHSR middleware layer, which takes care of the data packet handling, constructing packets for sending and dismantling received packets. The WHSR middleware also takes care of message transmission by utilizing the WHSR communication layer. There are two types of message frames regarding the communication. Push data messages are used for single direction communication, where the sender sends data to the receiver, but does not expect any response or acknowledgment. The second message type is request-reply, where the message sender is expecting a response or acknowledgement for the request message from the receiving end. (Wärtsilä, 2019b).

The WHSR communication layer is responsible for addressing and transfer of data packed by the middleware. Communication layer utilizes the WMAP platform hardware abstraction layer known as HAL to achieve the transmission and reception of messages. The communication messages are prioritized by using priority codes of the IEEE 802.1Q VLAN (Virtual Local Area Network) tag. Ethernet frames in the HSR network are sent as broadcast, and the message destination address is resolved on the communication layer. (Wärtsilä, 2019b)

3.3 Communication abstraction layer

The WHSR communication stack is utilized by the service handlers located in the Communication Abstraction Layer abbreviated as CAL, providing communication services for the different system components. Objective of the CAL is to abstract different communication protocols of WMAP from the different communication services. The layer consists in total of 11 service handlers. Out of the 11 service handlers, 7 utilize WHSR to transport the service specific data. The next subchapters will introduce the WHSR related service handlers, and how their service specific messages are sent over WHSR. After introducing all service handlers, Chapter 4 will discuss how the service handlers could utilize OPC-UA instead of WHSR. (Wärtsilä, 2019c)

3.3.1 Module status service handler

The purpose of module status service handler is to send and observe the incoming heartbeat messages. Heartbeat messages are periodically sent “still alive” messages indicating that the sending module is working properly and is available in the network. In WHSR the heartbeat is sent either in shorter or longer intervals, depending on the module status. A module is considered not available if no heartbeat message is received from it for certain amount of time. (Wärtsilä, 2021)

The heartbeat message header contains the status information of the sending module. The status information contains the state of the sender module and the module serial number. In addition to the module status information, the message contains status fields. Other service handlers and WMAP software components can utilize these status fields to distribute service specific status information. (Wärtsilä, 2017)

3.3.2 System command service handler

The system command service handler allows external control of software. The system command messages are request-reply type. A system command message header determines the message and command type, and the payload contains command specific arguments. There are in total 20 different system commands, which have their own payload data structures for request and reply. (Wärtsilä, 2017)

3.3.3 Enhanced diagnostic log service handler

Enhanced Diagnostic Log known as EDL is a logging system that logs important events happening on the engine such as mode changes and safety alarms. The purpose of the EDL service handler is to distribute and synchronize appearing log entries between the UNIC modules. The system has 5 different log types each logging the log type specific messages. The EDL message header defines the message type, and the EDL log type the message concerns. (Wärtsilä, 2017)

The EDL messages are distributed between modules with push type WHSR messages. The WHSR middleware can pack multiple EDL message packets containing message information on a single WHSR frame. EDL service handler also contains two EDL commands, which are request-reply type messages (Wärtsilä 2019d). The EDL commands are used for synchronizing the whole log history of the system, and to distinguish any lost messages. (Wärtsilä, 2017)

3.3.4 File transfer service handler

The file transfer service handler is used for transferring files between the UNIC modules. Transportation of a file is achieved with a transaction requiring multiple request-reply messages. File transfer utilizes frame splitting, meaning that an arbitrary size file can be transported in multiple chunks. The file transfer service handler provides progress information of an ongoing transfer, and it can detect if the transmission of an information chunk was successful. If transmission failure is detected the sender will automatically try to re-send the failed information chunk. Taking too many re-send attempts will cancel the transaction, indicating that there is an error in the system. (Wärtsilä, 2019c)

3.3.5 Synchronous parameter access

Synchronous Parameter Access abbreviated as SPA is used to externally read and write the current process values and parameters from the maintenance and monitoring tool (Wärtsilä, 2020a). SPA service handler uses a single request-reply type message to achieve read and write access for both parameter and process values. A message can contain multiple read or write transactions (Wärtsilä 2017)

3.3.6 Remote measurement service

Remote measurement service, abbreviated as RMS, is a service handler that provides subscription style external reading of the parameters and process values. Whereas SPA is used for infrequent reading, the RMS can be used when for example trending data

value changes over time by sampling value within certain intervals. For regular reading, the RMS gives improved performance over using SPA. (Wärtsilä, 2020b)

RMS service handler uses push type messages to publish data of a subscription, and request-reply messages are used for maintaining the subscriptions. When creating a subscription, in addition to the measured value the event for triggering the measurement is defined. Usually, the measurements are taken within certain time-intervals triggered by the system scheduler. (Wärtsilä, 2020b)

When subscribed to a measurement service, the data will periodically be sent from the module to the consumer according to the chosen event type. In these push type messages the header contains the event information, like timestamp when the measurements were sampled. The payload consists of multiple measurement information packets, containing the measurement ID, and the measurement data.

3.3.7 Data container distribution

The previously introduced SPA and RMS mainly consider the external access to the control system data, in which case responsibility of WHSR is to send or fetch data between the target module and the COM-10 gateway module that is taking care of the external communication. For distributing the measured and calculated process values and their statuses between the modules, the DC (Data Container) distribution service handler is used. The process value information is distributed with push type frames in the control system. Distribution logic i.e., which modules are the distributers, and which are the receivers, is pre-configured with an external software tool. (Wärtsilä, 2019b)

4 Service and protocol mappings

With the previous examinations it can be said that while OPC-UA is very general collection of protocols and focuses on information modelling and application level communication, the WHSR focuses just on transporting information and is designed to purely serve the UNIC control system, particularly the communication service handlers. OPC-UA considers mostly communication happening on the transport layer and above in terms of OSI-model, while WHSR is responsible only for the networking and transportation layers of the communication, and the application level is left for the CAL service handlers. On the other hand, the service handler features provided in WMAP are very similar to the OPC-UA features.

When considering the current system, applying OPC-UA would not only replace the WHSR protocol stack, but also some of the functionalities provided by the CAL service handlers. OPC-UA software automatically handles the received messages and modifies the address space accordingly.

One general mapping that can be made is that the push type messages of WHSR are similar to the publish messages of OPC-UA *PubSub*. Similarly, the request-reply type messages of WHSR correspond to the requests-response services of the client-server model. As a conclusion, UNIC would need both client-server and *PubSub* models to achieve similar communication with OPC-UA as with WHSR.

A new requirement that the use of OPC-UA brings is the information modelling concepts. To achieve data exchange with OPC-UA it is necessary to model the information in the OPC-UA server *AddressSpace* with the *Nodes* and *References* as described in Chapter 2. This will of course consume more resources, as there needs to be a separate representation of the parameter and process values in the OPC-UA *AddressSpace*, whereas with WHSR the current data structures are accessed as is.

The replacement of WHSR with OPC-UA needs to be considered on two levels. First, the CAL service handler abstractions need to be mapped to the OPC-UA services. This can be done by inspecting the service handler functionality and considering if the service logic itself could also be replaced, or if additional handling beside the OPC-UA services is required. Second, the transportation mappings for the communication needs to be considered. This means how the OPC-UA stack can interface towards the lower layers of the system (network and datalink layers of OSI) to achieve the message transmission. High-level overview of the concept of integrating OPC-UA to the WMAP communication stack is visualized in Figure 22.

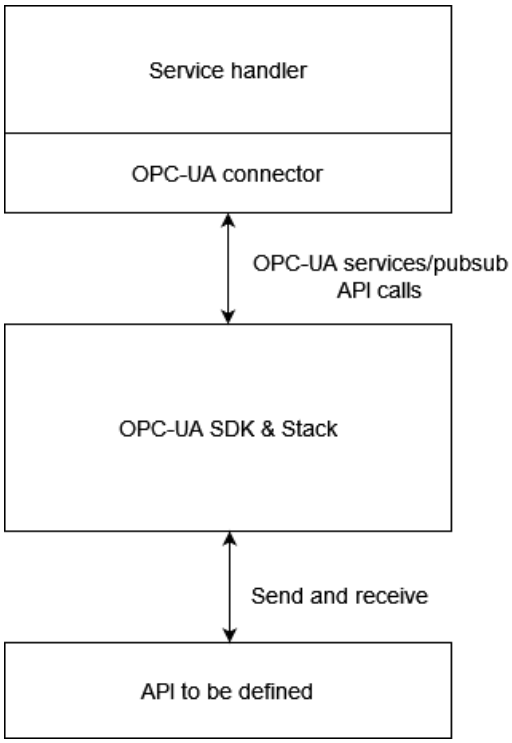

Figure 22. Concept overview for replacing WHSR with OPC-UA SDK.

4.1 Mapping service handlers

To create an OPC-UA connector for a service handler, the current WHSR based implementation can be mapped to the corresponding features of OPC-UA. The first thing to consider is should client-server or *PubSub* model be used. When the communication

model is chosen, it should be decided which features of the communication model should be used to achieve similar information exchange as with WHSR. In addition, it needs to be considered what information needs to be modelled to the OPC-UA *AddressSpace*, since information modelling is mandatory for achieving the data exchange.

Proposal for the service handler mappings is shown in Table 1. The first column of the table is the service to be mapped. Second column dictates if features of Client-Server or Pub-Sub shall be used, and the third column is the service to be used. Last column describes the minimum information that shall be modelled for utilizing the services. The following subchapters will discuss in more detail how the service handler services could be implemented with OPC-UA.

Table 1. Mapping of service handler services to OPC-UA features.

Service handler service	**Communication**	**OPC-UA feature**	**Information to model**
Module status service	*PubSub*	Publish and subscribe on module status *Event*.	*EventNotifier* information for transferring required *Event* information.
System commands	Client-Server	Call service	Commands as *Methods*. Optionally: related software components as *Objects* the commands relate to.
EDL distribution	*PubSub*	*PubSub* EDL messages as *Events*	EDL logs as *EventNotifiers*
EDL commands	Client-Server	Call service	EDL commands as *Methods*
File transfer	Client-Server	Call service with file transfer information model	The used file system
SPA	Client-Server	Read & Write	Parameter and process data
RMS	Client-Server	Subscription services	Parameter and process data
DC distribution	*PubSub*	*PubSub* with process values divided to published datasets	Process data

4.1.1 Module status service mapping

In general, there is no suitable heartbeat functionality in OPC-UA that would constantly monitor the availability of other network nodes. Although there are life pings for maintaining for example open sessions, the time-out periods of these features are a lot longer than the current failure detection time. As the module status is currently sent with push type messages, the *PubSub* mechanism could be considered as the correct communication model to implement this feature. Also, the status fields of the status message are a WHSR implementation specific functionality, so they shall be omitted from the OPC-UA implementation.

In theory, the heartbeat functionality could be implemented with a custom handler application on top of OPC-UA. An *EventNotifier* could be defined that the application could use to trigger heartbeat *Events* within certain intervals, which would be published with *PubSub* to other modules. The application would also subscribe for the heartbeat *Events* of other modules and track the available modules based on the received *Notifications*. In fact, following module status messages and determining the available modules are already the jobs of the service handler itself, so it could be utilized by implementing a connector for sending and receiving the heartbeat *Events*.

4.1.2 System command service mapping

As discussed in Chapter 3.3.2, the system commands are sent in request-response style, which would insist to use the client-server model. The system command purpose of controlling software execution could be mapped to *Methods* and the *Call services* of the client-server model. Each command type would correspond to a *Method* and the *Call service* could be used to call the *Methods* with the command specific input arguments and return possible result arguments.

Using *Methods* to implement system commands would also bring a challenge of modelling the *Methods*. As *Methods* are functionalities of *Objects*, it can be questioned how

the WMAP software components utilizing the commands should be modelled. In contrast, one could just ignore good modelling practices and collect all the command *Methods* into a single “Commands” *Object*. The simple approach is illustrated in Figure 23. The model includes also own input and output arguments for each method. In addition to modelling, each command *Method* needs to be assigned a callback function on the server that executed the command when the *Method* is called, as it was shown in Figure 14.

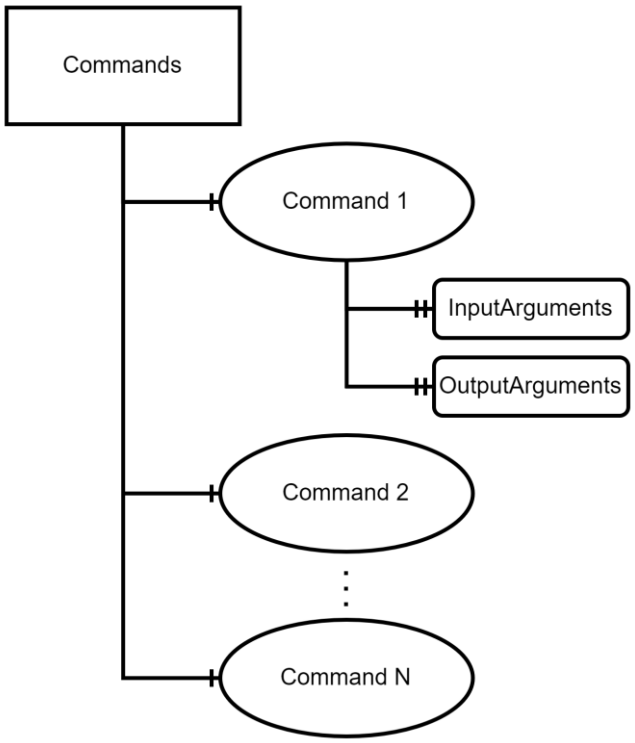

Figure 23. Modelling the system commands as *Methods* inside a “Commands” *Object*.

4.1.3 Enhanced Diagnostics Log service mapping

Currently the EDL services use request-response messages for retrieving old log entries and push messages to distribute the EDL data. EDL messages could be modelled as *Events* produced by an EDL log that could be modelled as *EventNotifier* in the *AddressSpace*. Following the message type pattern, the EDL distribution shall be implemented with *PubSub*, and EDL commands with client-server model.

To access the previous sequence numbers with services, the historical access features could be used. In case of reset command (request whole history), the whole event history could be fetched with history read without any filtering. History read could also be used to implement the request from sequence number feature, by specifying filter criteria for the read. A downside for using historical access, is that the log history would likely require some kind of database for managing the log history that could be accessed with the historical access services.

Another simpler approach for implementing the EDL commands would be to use *Methods* as with system commands. In this approach the EDL log would be accessed with the service handler application that would manage the log and return newer messages for a given a sequence number. Each of the log types could be modeless as an *Object* in the OPC-UA *AddressSpace* and have a *Method* where input would be the sequence number of interest, and output would be a set of EDL messages that are newer than the requested sequence number. Figure 24 illustrates the previously mentioned option.

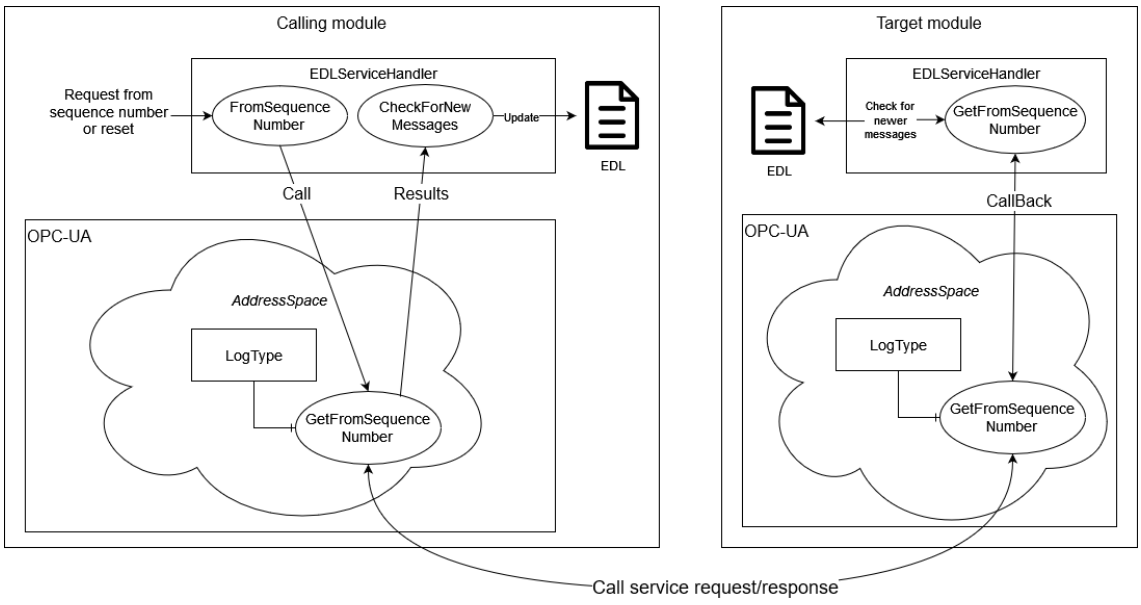

Figure 24. Simple approach for EDL commands with OPC-UA, with minimal information modelling.

4.1.4 File transfer service mapping

OPC-Foundation (2022f) has defined modelling rules for creating a file transfer infrastructure in Part 20 of the specification. The file system is modelled with directory *Objects* and file *Objects*. Creating and interacting with directories and files is done with *Methods*. For example, a file *Object* has *Methods* for opening, reading, writing, and closing. In principle the reading and writing happens in similar fashion to WHSR, where files are transferred in chunks by moving the offset inside the file. With this specification Part, implementing the file transfer services with OPC-UA should be straightforward. It just requires modelling the used filesystem parts and the files in the servers *AddressSpace* and mapping the *Methods* to the platform specific system calls. A modelling example of organizing files with a filesystem is shown in Figure 25, where on the left is the type definition for the file directory *ObjectType* and on the right is an example filesystem structure for a device.

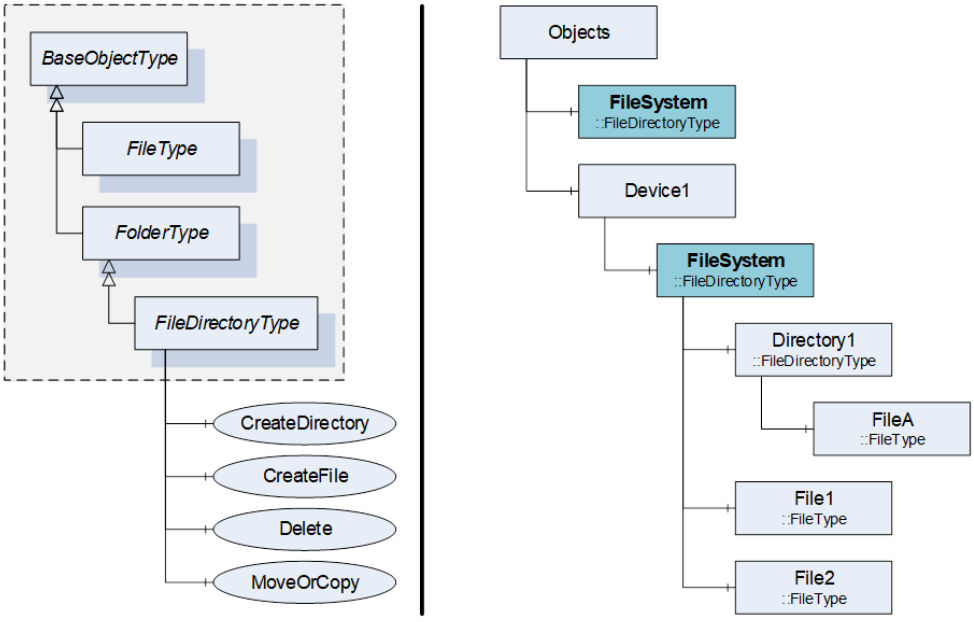

Figure 25. Modelling of a file system. On the left is the type hierarchy, and on the right an example of using the types to model the file system of a device. (OPC-Foundation, 2022f)

4.1.5 Synchronous parameter access mapping

SPA service can be mapped to the read and write services of OPC-UA, as both use very similar request-response style of reading and writing of attributes. The main thing to consider in this case is how the parameter and process value data structures should be modelled in the OPC-UA *AdressSpace*. As SPA considers delivering data between the modules and the external tool via the gateway module the OPC-UA version could be implemented so that each module implements a server that the gateway module accesses as client.

4.1.6 Remote measurement service

RMS publishes measurements in similar fashion with *PubSub* model of OPC-UA. One issue of implementing RMS with *PubSub* is that the *DataSets* are usually static while RMS subscriptions are modified during runtime. According to the *PubSub* specification (OPC-Foundation, 2022d) it is also possible to use client-server model services to modify the *DataSets* during operation, which is similar to the procedure how request-response is used with WHSR to subscribe to measurements. For configuring the publishers and subscribers with services, the *PubSub* specification defines a configuration model, where method calls can be used for configuring the *PubSub* components. (OPC-Foundation, 2022d)

Another option would be the *Subscription services* of client-server model. When comparing to the *PubSub* option the client-server *Subscriptions* seem to be significantly simpler for dynamically creating and modifying subscriptions as there are less details to model and configure. In addition, the MonitoredItems allow triggering items, which is similar to the non-scheduler triggered events used with WHSR. Again, simplicity comes with the cost of maintaining the *Session* between the target module, consuming more computing resources. An example of using the subscription service is shown in Figure 26.

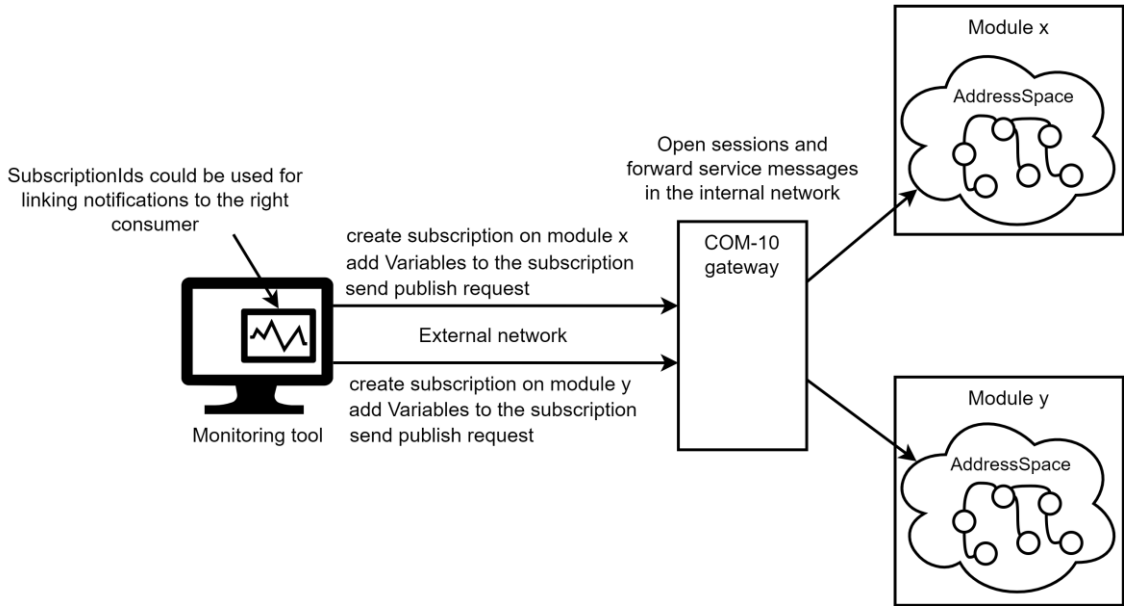

Figure 26. Measurement service by using OPC-UA Subscription services.

4.1.7 Data Container distribution mapping

For DC distribution service handler, the OPC-UA equivalent would certainly be the *Pub-Sub* model. Process values are distributed with push messages, and the data sets are fixed, as the frame templates are derived from the communication lists. One detail to keep in mind for implementation is that with *PubSub* also the subscriber needs to be configured to listen the messages, unlike in WHSR where messages are straight sent to the address of the receiver. The existing functionality of the communication lists could be used when creating the *PublishedDataSets* and *SubscribedDataSet* for the process values.

4.2 Transportation mapping

Issues arise, when OPC-UA transportation mappings need to be considered for UNIC. This is because UNIC uses real-time communication. The client-server model of OPC-UA is not real-time capable, as it can only be used over TCP/IP based protocols which do not achieve hard real-time requirements (Mirsky, 2008). Furthermore, the UNIC modules do

not currently use IP addressing. Although TCP/IP is not currently in use, there exists a TCP/IP stack in the network stack UNIC currently uses.

In contrast to the client-server model, the *PubSub* has an Ethernet transportation mapping beside the IP based options (OPC-Foundation, 2022d). This Ethernet mapping could be used to publish information in a real-time capable Ethernet network. The *PubSub* specification itself does not assume any real-time networking details of the Ethernet, it just shortly describes how the sent frames shall use EtherType code of 0xB62C and how the UADP *NetworkMessages* shall be inserted as the payload of the frame.

In theory the OPC-UA Ethernet mapping could be utilized with the same HAL services the WHSR implementation uses. Although the Ethernet frames sent only as broadcast with no knowledge of the module MAC-addresses (Media Access Control) the *Subscribers* could be configured to filter *DataSets* of interest from *NetworkMessages*. In other words, all data distributed with *PubSub* could be broadcasted to the HSR network and a *Subscriber* could filter module specific *DataSets*.

If disregarding the real-time incompatibility issues for a while, the OPC-UA transportation mappings to be used in UNIC are trivial. Ethernet mapping shall be used with *PubSub* to be real-time compatible, which also forces the use of UADP encoding (OPC-Foundation, 2022d). For client-server, the UA-TCP with binary encoding shall be used as it is the best performing option. For security, the most minimal security policies shall be chosen, as currently control system is well isolated from public networks and any excess security mechanisms would just consume resources.

5 Requirements for changing the protocol

From previous analysis on the service and transportation mappings it can be concluded that performing the tasks of WHSR with OPC-UA in the current UNIC system should be possible, although it would require significant amount of work to make the upgrade. To implement all services that currently utilize WHSR, a wide set of OPC-UA features is required with both Client-Server and *PubSub* models. This sets requirements for the used SDK to contain all the necessary features for successful implementation.

An issue that needs to be solved is the transportation of messages. As it was discussed in Chapter 4.2, the current Ethernet communication services provided by HAL are not sufficient for achieving communication with OPC-UA, as client-server requires TCP/IP communication. While currently all of the WHSR services are managed with the real-time Ethernet, it could be evaluated which of the services actually require the real-time communication. Services that do not require real-time determinism could then utilize the client-server model services, which most of the current service handlers would utilize according to the mapping made in Chapter 4.1.

5.1 Lifting real time requirements

While inspecting the service purposes and how the functionality is currently implemented, it can be argued that most of the service handlers mapped to client-server model do not require real-time determinism. File transfer is obviously not time critical as it is used on software download process which is not time sensitive. SPA could also be used over non-real-time network, as real-time determinism is usually measured in milliseconds, which is imperceptible comparing to the response time of a human operator. Neither should the RMS services require real-time precision. As measurements are time-stamped, the time to transport the values to the measurement tool does not affect the measurement results. Furthermore, SPA and RMS are already transferred over a non-deterministic network when moving between the gateway module to the external tool.

System commands seem to be used for executing functions that are not time critical. The time-criticality of the different system commands should be verified from experts knowing better the use cases for each command. The EDL commands can also be performed over non-deterministic transportation, as they are also currently handled asynchronously so that the possible missing sequence numbers are scheduled to a publish message rather than sending them in the reply. For simplicity, the concepts presented in this thesis assumes that all the services mapped to client-server model are not real-time critical.

5.2 Software development kit requirements

Based on the previous mappings the following base requirements can be set for the SDK that would be used for implementing the mapped services:

1. Implementations of client, server, publisher, and subscriber needs to be included.
2. There needs to be implementations for all required services listed in Table 1.
3. Mappings for UA Binary-UA TCP and UADP-Ethernet need to be included.

Currently, the WMAP platform integrates an OPC-UA SDK, that does not contain the *Pub-Sub* Ethernet mapping. Hence, the current SDK would need to be changed in order to implement both internal and external communication with the same SDK. Using two separate SDKs would be inefficient and cause a lot of interoperability issues, as SDKs use different naming conventions.

While searching for development kits fulfilling the previously set requirements, it seems that the Ethernet mapping is not currently widely supported, as it is absent from almost every SDK. Only a single SDK implementing the Ethernet mapping could be found during the search, that being the open62541 project. Currently the open62541 seems to fulfill all the requirements, except the file transfer model. Although there is currently no file transfer feature available out of the box in the open62541, a pull request draft implementing *FileType* could be found from the projects Github page (open62541, 2022). With

source code of this pull request, implementing the file transfer should be a moderate size task.

In Wärtsilä (2019e), there has also previously been evaluation between the different SDKs when the current SDK was chosen. From the evaluation matrix it can be interpreted that open62541 was already considered as a strong option to be in WMAP for external communication. This further promotes using open62541 for implementing internal and external OPC-UA communication on UNIC.

5.3 Network requirements

To open connections and establish transmission of messages, it is required from the platform software to provide interfaces for opening and managing TCP/IP connections, and for publishing network messages with real-time Ethernet. In addition to the TCP/IP functionality, there needs to be a mechanism that sets the IP addresses of the modules. The module IP address could be set on the module initialization phase according to the module hardware ID.

When thinking longer into the future for the next generation UNIC and its network implementation, the incoming IEC/IEEE 60802 standard could be considered. If the standard gets widely adopted, it will allow connecting UNIC modules with other IEC/IEEE 60802 compliant devices into same TSN time domain further enhancing interoperability. Although the IEC/IEEE 60802 project is not yet widely available, there are few standards that will surely be included. One of these is the IEEE 802.1AS, defining timing and synchronization mechanisms, which is an adaption profile of the PTP (Precision Time Protocol) defined in IEEE 1588, that is currently also used in UNIC.

Carlstedt, Liu, and Wang (2020) have published a white paper, where they test latency and jitter with OPC-UA *PubSub* over TSN. In the tests, in addition to IEEE 802.1AS, they used IEEE 802.1Qbv time-aware scheduling mechanism, and IEEE 802.1Qbu/802.3br frame preemption. The performance of these protocols was remarkable especially when

best effort traffic (non-deterministic) was sent simultaneously in the network with the real-time traffic. In this case the average jitter without TSN capabilities was 85 microseconds, while with both time aware scheduling and frame preemption active, the average jitter was only 76 nanoseconds. Similar testing has also been performed by Pfrommer et al. (2018) where IEEE 802.1Qbv based scheduling was used. Their results were very similar to Carlstedt, Liu, and Wang (2020), where average jitter was reduced from the scale of microseconds to nanoseconds.

In conclusion, although the TSN profile for industrial automation is not yet published, the standards that will be contained in the profile are already established. After the IEC/IEEE 60802 has been published, it will be interesting how the TSN will be handled in the upcoming OPC-UA FLC specifications. This thesis will not go any deeper on investigating the different TSN protocols, as they are not directly part of the OPC-UA technology. What OPC-UA concerns about the protocols is that how they can be configured using the information modelling concepts.

6 Communication architecture proposal

As OPC-UA is very wide standard and different SKDs having their own APIs, this thesis presents only a high-level overview how the current services handled with WHSR could be implemented with OPC-UA. The architecture discussion is divided into two parts. The first part considers the application software architecture i.e., what software components each of the modules should have, how their internal structure could be laid out, and how they could connect to the existing service handlers. The second part proposes an information model concept for the modules. The proposed information model begins with a minimum effort model and further discusses how the information model could be enhanced.

6.1 Software component architecture

The diagram shown in Figure 27 considers how the client, server, publisher, and subscriber interfaces should be implemented on different modules. The client-server communication could be handled so that every module implements a server, which is accessed with a single client located in the COM-10 gateway module. This is because all the WHSR services mapped to the client-server model consider services utilized from the external systems. Modules themselves do not use utilize these services. The request is always sent from the gateway to a target module. In addition to the gateway module, other COM-10 modules would likely require a client interface for using just EDL commands to maintain their logs.

For *PubSub*, each of the modules would implement both a *Publisher* and a *Subscriber* interface. A *Publisher* would broadcast EDL messages, process value data, and the module status *Event* to the network. The *Subscribers* need be configured to listen certain *DataSetMessages*. For example, only the designated COM-10 modules maintaining all EDL messages would be required to subscribe for the sets containing EDL messages.

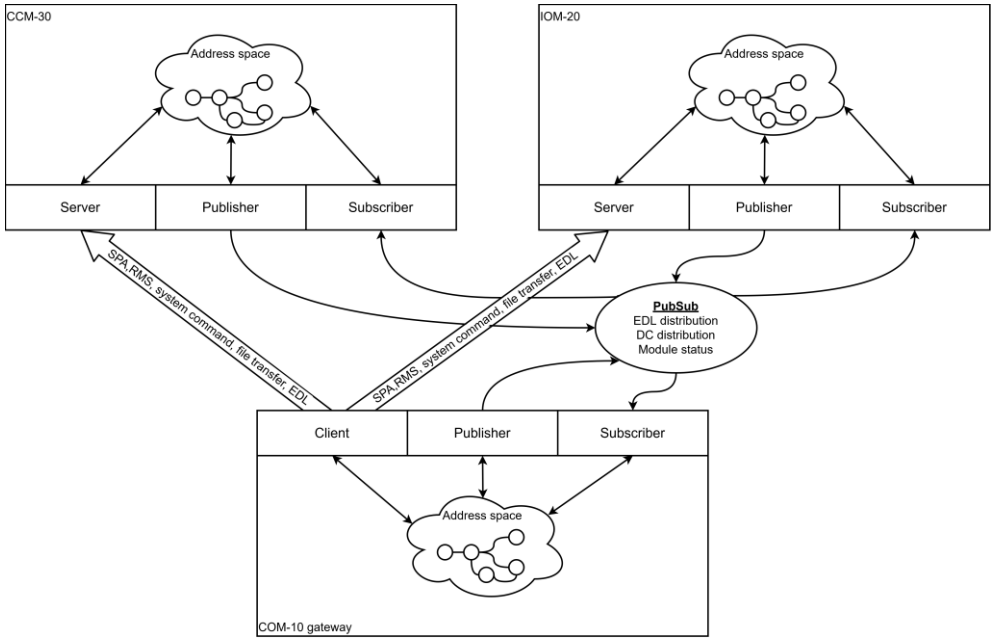

Figure 27. Architecture diagram showing which application interfaces the different UNIC modules should implement, and what services are they handling.

6.1.1 Common server initialization

Initialization steps of the common OPC-UA interface with a server, publisher, and subscriber is visualized in Figure 28. The concept follows details of the open62541 SDK, where *PubSub connector* and *AddressSpace* are encapsulated inside the server component. The inner structure of other SDKs can vary, but essentially same initialization steps are required for setting up the communication.

The initialization steps could be implemented in a “OPC-UA Initializer” file that the system could use in the UNIC initialization phase. The process would start by initializing and configuring the server details. A pre-requisite for initialization is resolving the module IP-addressing so that the addresses can be used when defining the endpoint URL (Uniform Resource Locator) of the server, which the OPC-UA client uses for connecting to the server. Otherwise, the configuration is mostly about setting limits for different components and enabling/disabling features. It shall be also noted that inclusion of different SDK components like *PubSub* is usually done during compile time and therefore need to be addressed by the build system.

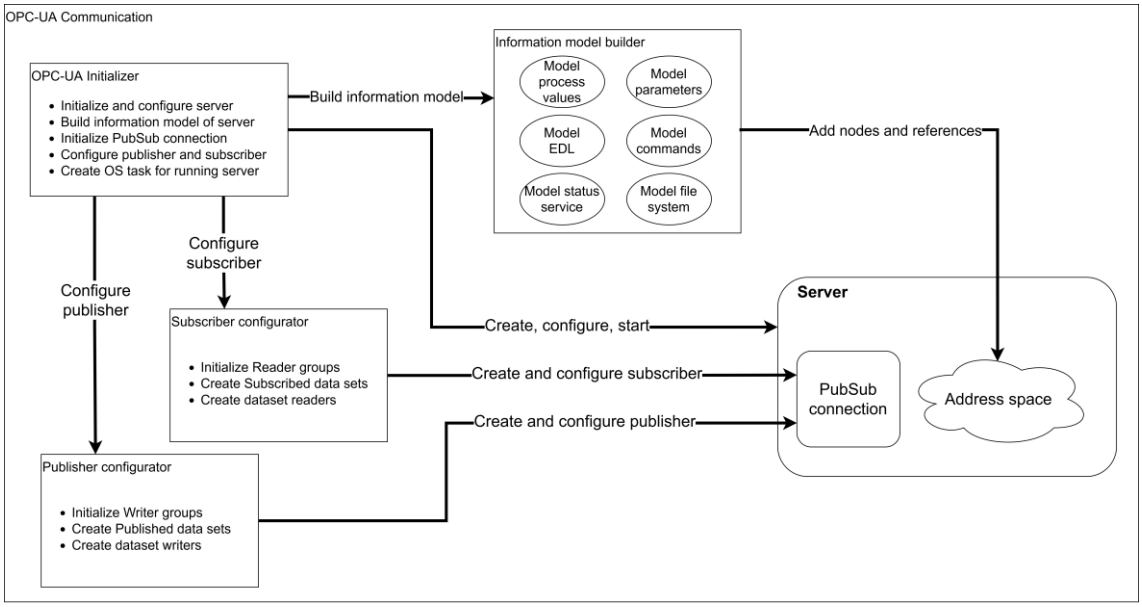

Figure 28. Initializing the common OPC-UA communication components on modules.

After the server base is initialized and configured, the required information models need to be added into it. For this purpose, it would be wise to perform all the information modelling in a separate c-file that would have own functions for building the different features to the information model. Essentially what is done in the functions is creating different types of *Nodes* and adding them to the server address space. For example, when process values are modelled, a process value *ObjectType* can be defined and used to create process values as *Objects* with *Variables* representing the different attributes. The data structure containing the process value symbols could be iterated through with a simple loop, creating the required *Nodes* for each code.

After the information models are added, the *Publisher* and *Subscriber* interfaces can be added to the server. In case of open62541 (2021b) SDK, *PubSub* connection holding the *ReaderGroups* and *WriterGroups* would be added on the server component. To these groups can then be added the *PublishedDataSets* and *SubscribedDataSets* by creating *DataSetWriters* and *DataSetReaders*. Again, the *Publisher* and *Subscriber* configurations could be separated to their own files, as they require some logic for deciding what needs to be published or subscribed.

When everything is configured to the address space, the server can be run. This is again SDK dependant. In open62541 (2021b) there are options for using a built-in loop to continuously update the server or calling a function to iterate a single cycle. In contrast, the currently used SDK has a mechanism that allows to define time window for how long the server code shall iterate before returning. This type of iteration window could also be implemented with open62541 by iterating the server inside a custom while loop.

When a server task is created, the server should run cyclically, and the *Publisher* publishes data cyclically, which the *Subscribers* listen to. Also, the server should automatically respond to the requests of a client based on the information model in the *AddressSpace*. To keep the data in the *AddressSpace* updated, there needs to be a mechanism connecting the *AddressSpace* values to the real measurement and control values. In open62541 (2021b) documentation a few options are introduced. A common method is to assign a callback function for a *Variable* that when reading the value is first copied from its source before sending, and when writing the value is copied from the information model to the physical destination after the writing operation has finished. These callbacks should be defined already during the modelling phase.

6.1.2 Client structure

In addition to the common server-publisher-subscriber component, the COM-10 gateway module should implement a client interface that can request services from other modules. Some of the service requests could be accessed via the service handler abstracted interfaces, but some of the abstraction layer interfaces could be completely changed, as they provide functionalities that are handled internally in OPC-UA. Integration of the client with the current service handlers is shown in Figure 29.

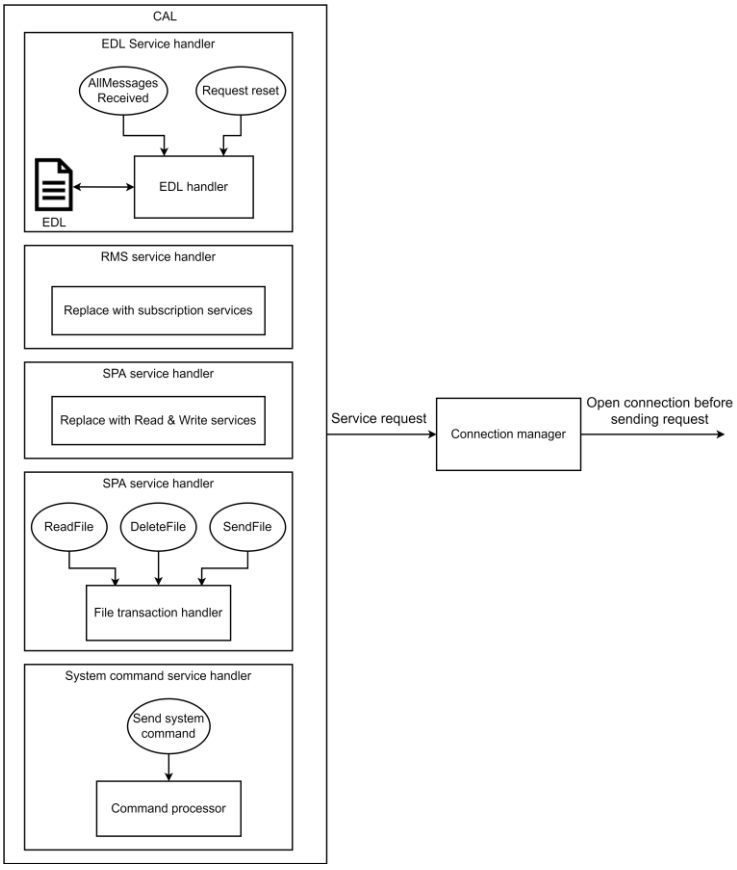

Figure 29. Accessing services via service handlers. A common connection manager software component ensures that an open session is available when any of the service handlers wants to send a service request.

To handle the EDL commands, the service handler needs functionality for accessing the EDL for checking the sequence number and to write any message received in responses. The handler would then map the request for the right *Method* call. Both commands could have their own *Methods* which would increase transparency.

Before the service requests can be sent, a communication session needs to be established. For this purpose, the client could have a separate session manager that ensures a session is opened before sending the request. It is likely the best practice to keep all sessions open by default and re-establish the connection before sending requests if a timeout has occurred. It should be further investigated how the open sessions affect performance versus continuously opening and closing session.

The file transfer service handler needs to control the transactions, with the *Methods* given in the file transfer specification, as on the abstraction layer the file is read or written with a single function. The *Methods* need to be connected to callback functions on the server side for processing the *Methods* with system specific calls. On client side the handler needs to process the response messages.

System commands need to also be processed so that the current command enumerations can be connected to the right *Method Nodes*. On the server side the service handler assigns *Methods* with callback functions that executes the actual functionality with system calls and returns the result. The command responses need to also be processed on the client side.

The RMS and SPA service handlers would require a complete overhaul, as currently they are just handling the requests on the target module and forwarding the messages between HSR and external network on the gateway module. If the current WHSR implementation is changed, it will break the current connector mechanism. When OPC-UA is considered for internal communication, it could be assumed that also the external would be performed with OPC-UA. If for some reason external communication would not use OPC-UA, some kind of adapter needs to be created that translates the messages to OPC-UA requests. The next chapter introduces a concept how the *AddressSpaces* of different modules could be aggregated and exposed to external systems.

6.1.3 Aggregating server

In UNIC each of the hardware modules have their own data instances of the control system variables, so it must be possible to select the module, where a *Variable* is read from. This is especially important in cases when a value is not distributed in the system. Accessing the *AddressSpaces* of different modules could be achieved by modelling each of the modules separately with their own instances of common data. This sets a problem, that how the modules would be able to exchange data, but at the same time the data could be accessed module wise.

A solution for the problem would be a so-called aggregating server. Seilonen et. al. (2016) define that an aggregating server is a server that provides services by combining services from a set of source servers. The information model of each module would just include data instances existing on that module and the aggregating server would combine the different server information models and redirect any service requests to the corresponding server. The aggregating server concept is visualized in Figure 30. While implementing the aggregating server, the gateway module should also implement its own server containing the module specific data instances, which the aggregating server can access when data is specifically requested from the gateway module.

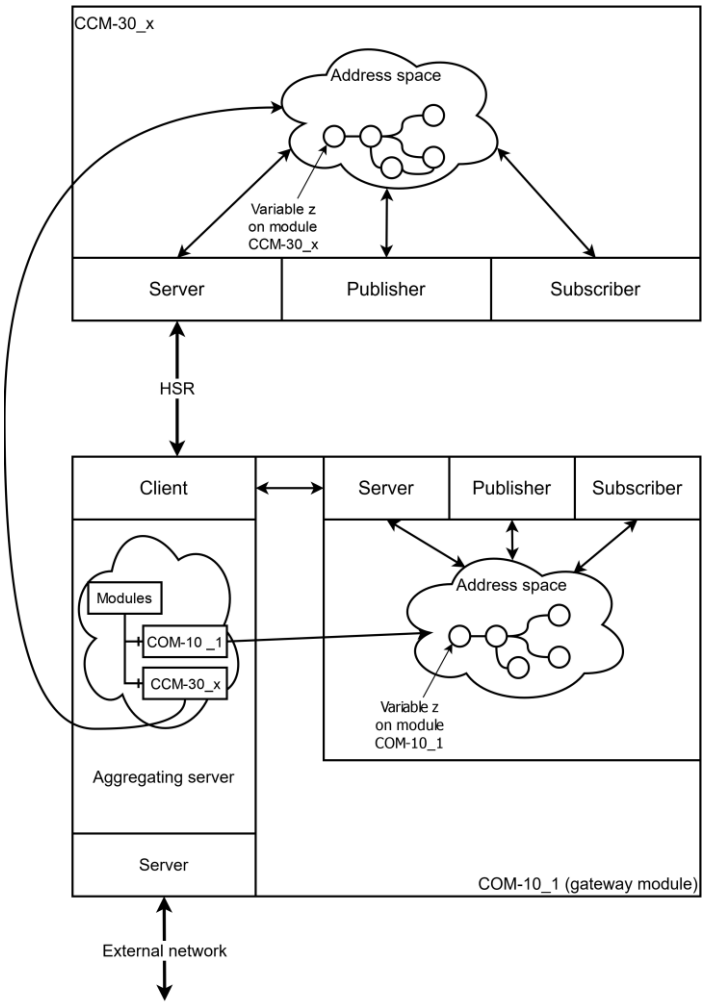

Figure 30. Accessing different module *AddressSpaces* via aggregating server.

Now the *AddressSpace* of the aggregating server can expose all information models of the UNIC modules separately and re-direct the service requests to the correct server. Simple in theory but requires some effort in practice to configure the aggregating *AddressSpace* accordingly with varying hardware module setup between different engine configurations. In addition to aggregating servers of the UNIC modules, the aggregating server could also aggregate the servers of other devices, if such are chosen to be connected into the network.

6.2 Information model structure

Information modelling could be started by thinking about the absolute minimum modelling, that would be required for exchanging necessary information. The proposed information model with minimal modelling is illustrated in Figures 31 and 32. Figure 31 shows the parameter and process value structures, and the system command part of the model. Figure 32 shows the EDL, file transfer, and module status parts of the model. The information model of a module would be contained in a module *Object*, which the aggregating server would aggregate inside a modules folder.

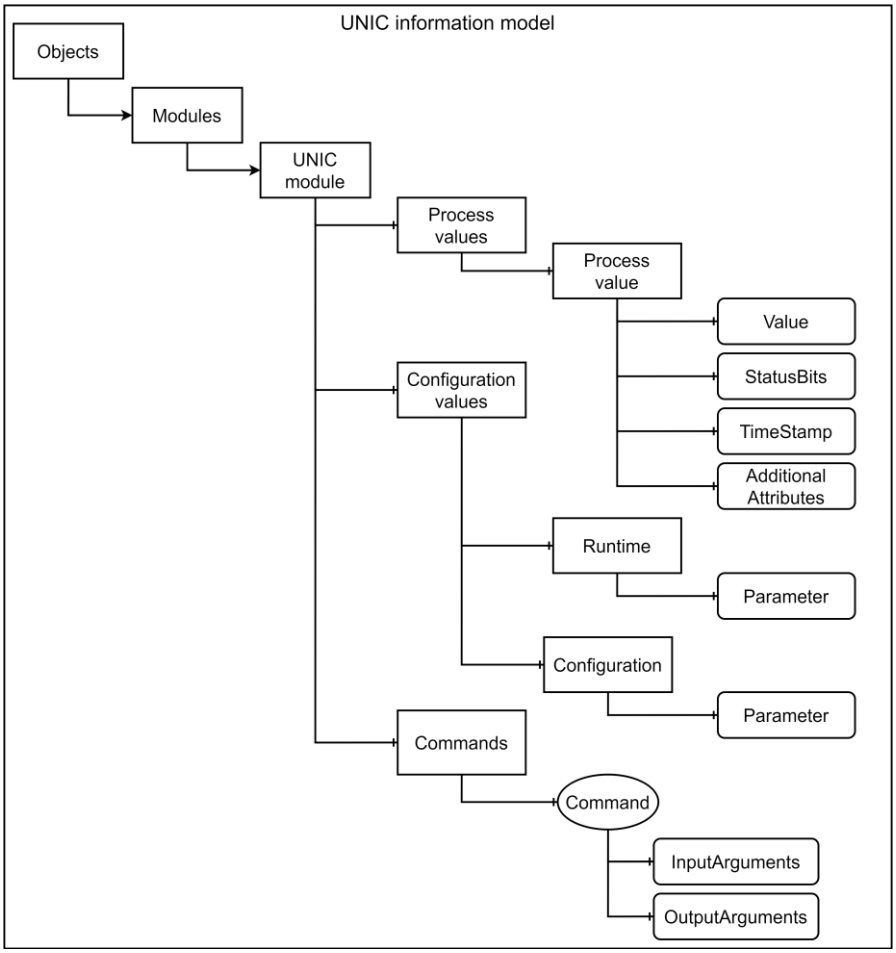

Figure 31. Proposed information model structure where process values, configuration values, and system commands are modelled inside a UNIC module.

The information model of a module consists of software components mentioned in Table 1. Process values would be modelled as *Objects* inside a folder *Object*. A process value *Object* consists of mandatory and optional attributes modelled with variables. The parameters can be separated into runtime and configuration structs containing the certain types of parameters. For system commands, as described in Chapter 4.1.2, the module could just simply have a commands *Object* containing the different command *Methods*.

With EDL, each of the log types could be an *Object*, having *Methods* for handling the log as described in Chapter 4.1.3. The EDL log would contain an *EventNotifier* that creates and triggers *Events* from new log messages. The *Event* information would contain the information of the log specific message.

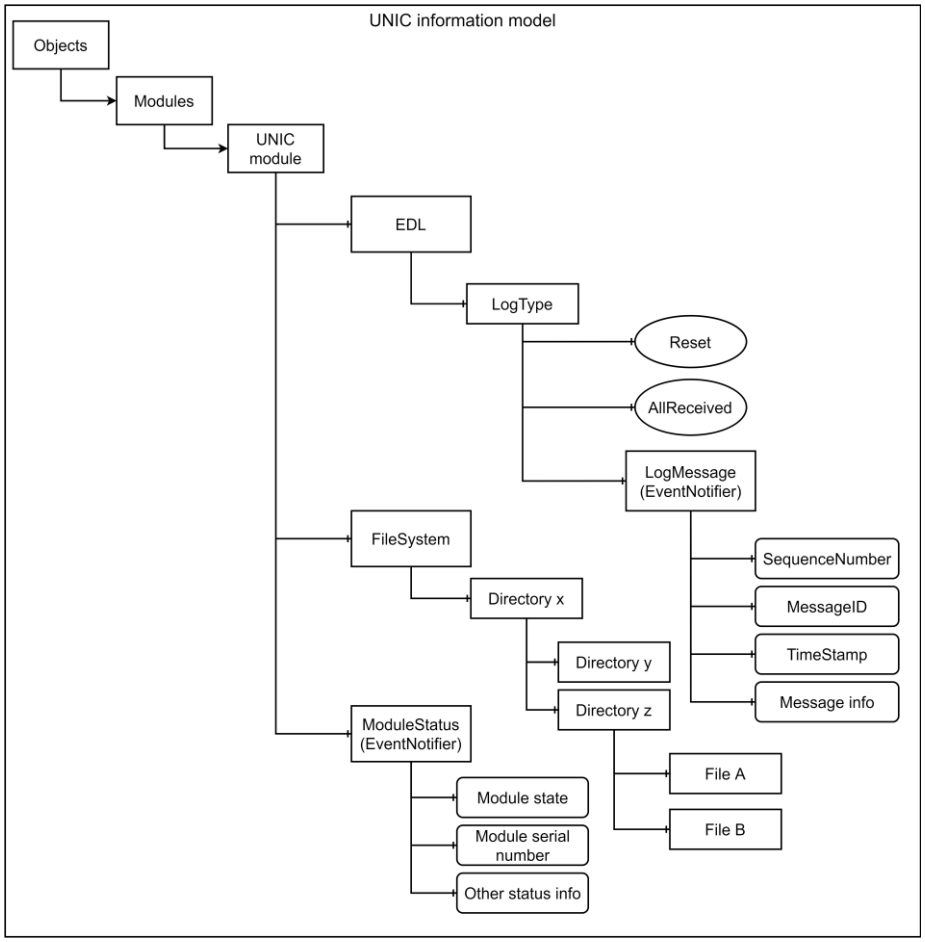

Figure 32. Information model structures for EDL, file system, and module status.

The file system of the module should be modelled with the directory and folder types described in Chapter 4.1.4. It should be enough to only model the directories that are used. In addition, *Methods* and their call-backs should also be implemented so that no unallowed actions are performed. For example, read-only files should not have a *Method* for writing.

Finally, a module has the module status *EventNotifier* for sending the module status *Event* as described in Chapter 4.1.1. The module status event should contain the same information as in WHSR status message. The *Events* would be triggered and listened by the service handler application which also follows statuses of other modules.

As model concept shown in Figures 31 and 32 implement bare minimum modelling for transferring necessary information, it does not utilize the full potential of OPC-UA, which is designed to expose rich information models described with metadata. Currently, a lot of the metadata is handled by the UNITool monitoring software. For example, the WMAP handlers accessing parameter or process values have no information about the name or unit of the value that it accesses. With OPC-UA the value structures could be modelled, to expose different attributes such as names, units, value ranges, etc. to increase transparency.

In addition to adding metadata about value information, the OPC-UA information modelling should be used for what it is designed for, modelling real world objects. In big picture, the challenge would be how to model an engine and its components. One starting point would be connecting the process values to the real signals moving around the engine. It can be examined, what are the sources or destinations of a signal and what is the physical engine part the signal concerns.

An example of an information model with the previously mentioned enhancements is shown in Figure 33. Now, the aggregating server would also model the physical systems of the engine, beside the previously aggregated UNIC modules, which could be modelled under the “UNIC control system” *Object*. Dividing the engine parts into smaller pieces could start by following the different systems that are currently used with the process value naming conventions (100 for fuel system, 200 for lube oil system, and so on). After that the different systems could be broken into smaller components until the level of sensors and actuators is reached. In the example the cooling water system has a low temperature water circuit, which has temperature sensors in different positions. Engine outlet temperature measurement is contained in the TE452 process value code. In addition, the value is now appended with some metadata describing the temperature value.

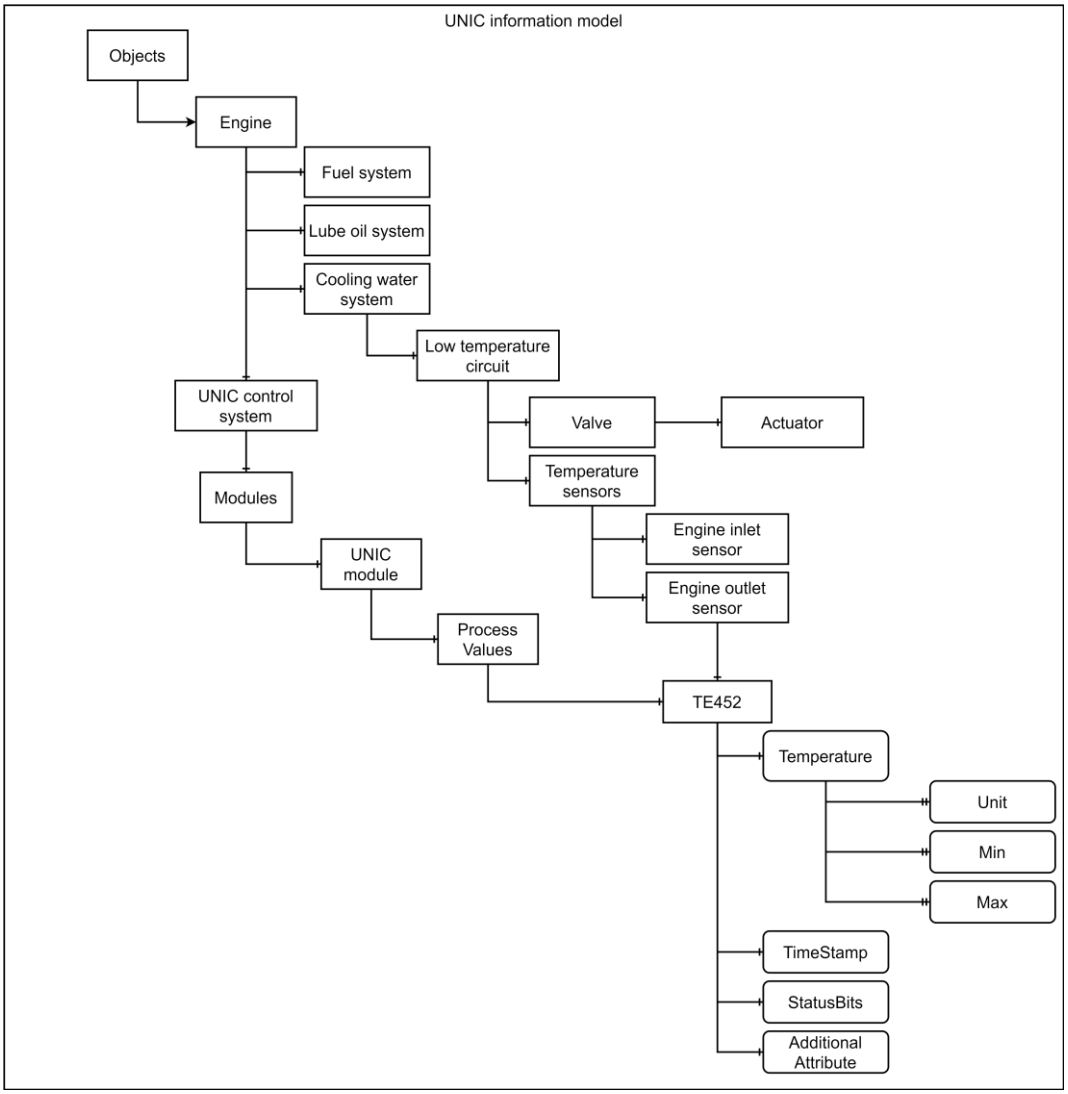

Figure 33. Model with metadata about the process values and the engine environment.

Building an information model with a lot of metadata can get complicated, so a set of modelling rules should be defined for keeping models consistent. An issue that essentially needs to be solved is how the model is created, as the engine structure varies between different engine types. One solution would be to create a nodeset XML file during the engine configuration phase, that could be parsed during initialization to build the model. If possible, the needed information could also be extracted from the actual engine configuration XML. The open62541 project has a compiler tool that can be used to create a server from an OPC-UA nodeset XML information (open62541, 2021b).

Designing a metadata rich information model for an engine is not an easy task, as already the number of data variables could be counted in thousands. Furthermore, presenting an engine with OPC-UA information model would likely be something that also other departments of Wärtsilä would be interested on. The engine modelling practices should be standardized somehow inside Wärtsilä to increase future interoperability.

When designing the data model, it would be wise to check the types provided by the OPC-UA core information models and companion specification instead of straight creating custom types. For example, the data access specification defines types for different data variables, such as *AnalogItemType*. Furthermore, it could be considered how for example EDL log messages could be converted to the event models presented in alarms and conditions specification. On the companion specification side, there currently exist no specifications that would explicitly define models for combustion engines, but there exists for example a specification for modelling pumps, that could be used to model the different pumps used on the engine.

7 Performance comparison

Although WHSR service handlers can be implemented with OPC-UA in multiple ways, the main concern is how the implementation would affect the systems performance. Without benchmarking a real implementation, some rough estimates can be made by comparing the frame structures that are sent on the wire. The comparison focuses on the payload of the Ethernet frame, as the Ethernet frame itself can be considered to stay the same. For OPC-UA the frame content is different for *PubSub*, and client-server, and depends on selected features such as security. For simplicity, the OPC-UA frame sizes are considered so that they would contain similar information as WHSR.

7.1 Comparing publish-subscribe model

With WHSR the common part of a message consists of 24 bytes. In OPC-UA *PubSub* the UADP *NetworkMessage* can be considered as the common frame part. The UADP *NetworkMessage* has a lot of features that are optional such as publisher information and timestamps. Theoretically, the network message could only have a single byte header if all optional fields would be disabled. With timestamp, and group header, the header would require around 20 bytes, depending on the configuration. (OPC-Foundation, 2022d)

When considering DC distribution, the service frame header contains 16 bytes of data, and the process value information blocks with at least 16 bytes each. With *PubSub* the DataSetmessage header features are similarly customizable as with network messages. To achieve same features as with WHSR, an estimate of 20-byte header can be made for a single *DataSetMessage* packed inside a network message. Also, the payload can be considered to have same length as WHSR process value information blocks, as only data of the *DataField* is sent. In conclusion, the frame size of WHSR push messages and OPC-UA network messages is roughly the same. (OPC-Foundation, 2022d)

One important feature of OPC-UA *PubSub* is the *KeyFrame-DeltaFrame* concept, where only fields with value changes are sent on *DeltaFrames* (OPC-Foundation, 2022d). *KeyFrames* contain the whole *DataSet* and can be sent cyclically after certain number of *DeltaFrames*. This would be more efficient than sending constantly all *DataFields*, like it is done with WHSR. On the other hand, WHSR implementation of DC distribution is highly optimized, as values are accessed straight from the HAL level transmission buffers rather copying the frames from the stream buffers.

7.2 Comparing client-server model

With OPC-UA client-server model, the OPC-UA messages are wrapped in IP and TCP packets, both requiring a minimum of 20 bytes each (Impreva n.d.; Beyah, Holloway & Copeland, 2002). Inside the ordinary TCP packet is the UA SC message, that contains the request or response. In combination, the message header, security header, and sequence header require 24 bytes when a message is sent on established *Session*. In addition, an *ExtensionObject* is added determining the message type and encoding (OPC-Foundation 2022c). A request payload is built from the common request header and the service specific request details. The common request header is around 40 bytes long depending on the used features. A read service request header contains 16 bytes, and the *ReadValueId* specifying the information to read requires around 18 bytes of information depending on for example the used *NodeId* type. In total, a frame for reading a single variable value requires well over 100 bytes.

The open62541 (2021b) documentation shows a Wireshark screen capture of read request for 4 variables. From the Wireshark capture it can be calculated that from the end of the Ethernet header to the end of the first *ReadValueId* the read request requires 142 bytes. The operation is a lot heavier than the current WHSR SPA implementation, which would just require 52 bytes for sending a read request for a single parameter. Figure 34 illustrates the frame size comparison, where the frame blocks are roughly in scale.

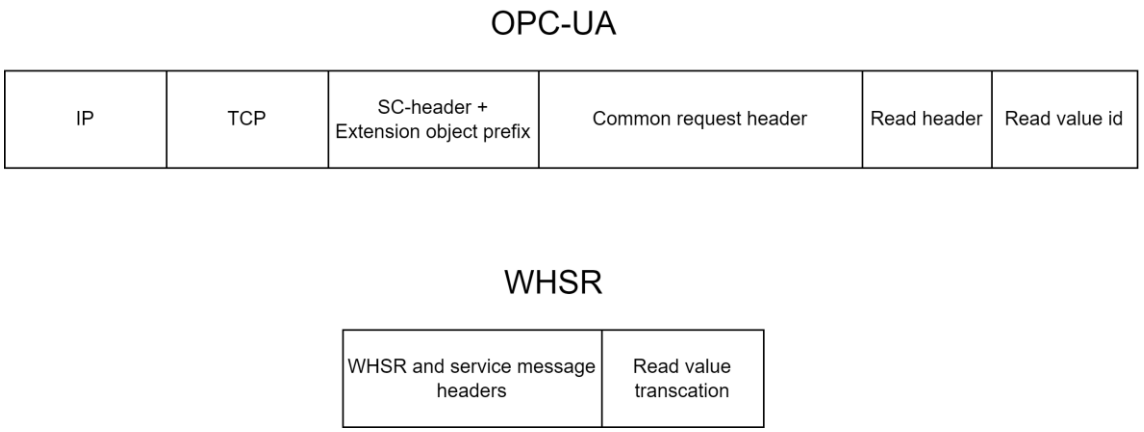

Figure 34. Scale comparison of WHSR and OPC-UA binary encoded read request frames.

Although the client-server request headers are a lot larger than the headers used in WHSR, the WHSR transaction and *ReadValueId* are similar in length. Hence, when more values are processed on a single request, the header overhead becomes less significant. It can also be considered that the client-server is used only for infrequent operations and therefore the message overhead is not highly important when compared to cyclical *Pub-Sub* distribution.

8 Conclusions

As a summary, the OPC-UA standard contains a wide variety of features that are similar to the communication services currently provided in the UNIC control system. OPC-UA also supports real-time communication requirements of the control system with the Ethernet mapping of the *PubSub* model. In addition, OPC-UA provides a non-deterministic client-server communication model over TCP/IP, which could be used for implementing the services utilizing the request-reply type WHSR messages.

As a conclusion, it can be said that OPC-UA has a high potential for replacing WHSR as the internal communication mechanism of UNIC, especially if it is also used for the external communication. This thesis proposed a concept how the UNIC system could externally be accessed via an aggregating server, that allows accessing the module specific data instances of process values.

A new aspect that needs to be considered when implementing OPC-UA is information modelling, the backbone of the OPC-UA communication. In principle, achieving internal data exchange does not require much modelling and all *Variables*, *Methods*, etc. could be laid horizontally into a single folder, but this would not be the best practice. Although providing metadata about the control system and the engine environment would be more important for the external communication, it should also be considered in the internal communication to improve the interoperability between the internal and external interfaces.

When comparing overhead of OPC-UA messages to WHSR, there are no obvious performance bottlenecks, that would suggest abandoning the OPC-UA communication concept. Although the client-server messages consume significantly more bandwidth, than WHSR request-reply messages, the performance on this type of messages should not be critical, as they are not being used continuously. However, performance testing would be required to discover how much the OPC-UA software components would require computing resources from the modules. Especially interesting would be the performance of the

gateway module, as it needs to manage multiple communication sessions with its client interface.

Although the features of OPC-UA standard cover a lot of the functionalities of current WHSR implementation, the technology is relatively new and there is limited amount of viable SDK solutions on the markets. Particularly, the *PubSub* real-time communication seems to be still in its experimental phase with very limited number of available solutions.

From economical perspective, replacing WHSR in the current UNIC2 control system is also somewhat questionable. Changing the whole internal communication stack to use OPC-UA would require a lot of working hours as on top of the implementation other codebases such as virtual test environments need to be also updated. The amount of modified source code lines would be counted in tens of thousands, and as a result no much new functionality would be added.

In the end, it can be stated that OPC-UA would be feasible for implementing the internal communication of the UNIC control system, but at its current state the technology is bit immature to be used in the control system, and the business value for the internal protocol upgrade would be low. After all, the benefits of OPC-UA target more the external communication of the system. The value what OPC-UA could bring to the internal communication would be to unify the internal and external communications rather than using separate protocols.

8.1 Future considerations

Although the real-time capable OPC-UA communication has not been widely adopted during the time of this thesis work, the technology is constantly advancing and should definitely be followed for the next few years. It will be interesting to see what the upcoming OPC-UA FLC standards define about the transportation over TSN, and how it will affect the adaptation of OPC-UA on the field level communication. It would be wise to also consider the current HSR network implementation, and how the TSN standards from

the IEC/IEEE 60802 industrial automation profile could be a viable upgrade for the system.

The concepts presented in this thesis could be more applicable when considering next generation control system for Wärtsilä engines. Instead of asking how OPC-UA could serve the custom features of the WMAP platform, the OPC-UA concepts could be used as guideline for the future system platform architecture, and the used data structures. Starting to build the future communication concept from an empty board would be wise, as the OPC-UA cancels out many of the current service handler features, making it difficult to maintain the backwards interoperability with WHSR.

In the end, performing communication of UNIC control system with OPC-UA would require a big leap from the control system department. To fully leverage the benefits of OPC-UA, the technology should also be considered companywide for building a common communication interface for connecting the engine control system with other systems to stride towards the benefits of cyber physical systems of Industry 4.0.

References

Aro, J. (2021). *OPC UA PubSub Explained*. ProsysOPC. Retrieved 31.1.2022 from https://www.prosysopc.com/blog/opc-ua-pubsub-explained/

Beyah, R. A., Holloway, M. C. & Copeland, J. A. (2002). Invisible Trojan: an architecture, implementation and detection method. The 2002 45th Midwest Symposium on Circuits and Systems, Tulsa, OK, USA, 4-7.8.2002 DOI:10.1109/MWSCAS.2002.1187083.

Carlstedt, M, Yabing, L. & Wang T. (2020). *OPC UA PubSub over TSN*. Retrieved 13.5.2022 from https://github.com/open62541/open62541/blob/master/examples/pub-sub_realtime/opc-ua-tsn-wrs.pdf

Cupek, R., Ziebinski, A. & Drewniak, M. (2017). An OPC UA server as a gateway that shares CAN network data and engineering knowledge. *2017 IEEE International Conference on Industrial Technology (ICIT),* Toronto, ON, Canada, 22-25.3.2017. DOI: 10.1109/ICIT.2017.7915574

GE Digital. (2022). About OPC UA HDA. Retrieved 2.6.2022 from https://www.ge.com/digital/documentation/historian/version2022/c_about_opc_ua_hda.html

Hoppe, S. (2017). *There Is No Industrie 4.0 without OPC UA.* OPC-Foundation. https://opcconnect.opcfoundation.org/2017/06/there-is-no-industrie-4-0-without-opc-ua/

Hoppe, S. & Stark, A. (2019). IoT Basics: What is OPC UA? Spotlightmetal. Retreived 28.3.2022 from https://www.spotlightmetal.com/iot-basics-what-is-opc-ua-a-842878/

Industry40tv. (2020). *Understanding OPC UA Base Information Model and Companion Specifications [3 of 11]* [video]. YouTube. Retrieved 16.1.2022 from https://youtu.be/cL5Tq7a1gwo

Industry40tv. (2021). *How OPC UA Client Server Communication works [5 of 11]* [video]. YouTube. Retrieved 18.1.2022 from https://youtu.be/vgE9P6KNC7g

Impreva. (n.d). *Transmission control protocol (TCP)*. Retrieved 10.6.2022 from https://www.imperva.com/learn/ddos/tcp-transmission-control-protocol/

Joung, J. & Kwon, J. (2021). Zero Jitter for Deterministic Networks Without Time-Synchronization. IEEE Access, vol. 9, 49398-49414, 2021, DOI:10.1109/ACCESS.2021.3068515.

Mahnke, W., Leitner, S. & Damm, M. (2009). *OPC unified architecture*. Springer.

Mirsky, S. (2008). *TCP/IP Communication for Real-Time and Embedded Systems.* Retrieved 6.5.2022 from https://se.mathworks.com/company/newsletters/articles/tcpip-communication-for-real-time-and-embedded-systems.html

OPC-Foundation. (2017a). *OPC Unified Architecture Part 1: Overview and Concepts* (Release 1.04). (OPC 10000-1). OPC-Foundation.

OPC-Foundation. (2021a). *OPC UA Field Level Communications (FLC) news.* Retrieved 3.5.2022 from https://opcconnect.opcfoundation.org/2021/12/opc-ua-field-level-communications-flc-news/

OPC-Foundation. (2021b). *OPC Unified Architecture Part 4: Services* (Release 1.05.00). (OPC 10000-4). OPC-Foundation.

OPC-Foundation. (2021c). Extending OPC UA to the field: OPC UA for Field eXchange (FX). Retrieved 30.5.2022 from https://opcfoundation.org/wp-content/uploads/2020/11/OPCF-FLC-Technical-Paper-C2C.pdf

OPC-Foundation. (2022a). *Unified Architecture*. OPC-Foundation. Retrieved 16.1.2022 from https://opcfoundation.org/about/opc-technologies/opc-ua/

OPC-Foundation. (2022b). *OPC Unified Architecture Part 3: Address Space Model* (Release 1.05.01). (OPC 10000-3). OPC-Foundation.

OPC-Foundation. (2022c). *OPC Unified Architecture Part 6: Mappings* (Release 1.05.01). (OPC 10000-6). OPC-Foundation.

OPC-Foundation. (2022d). *OPC Unified Architecture Part 14: PubSub* (Release 1.05.01). (OPC 10000-14). OPC-Foundation.

OPC-Foundation. (2022e). *OPC Unified Architecture Part 22: Base Network Model* (Release 1.05.01). (OPC 10000-22). OPC-Foundation.

OPC-Foundation. (2022f). *OPC Unified Architecture Part 20: File Transfer* (Release 1.05.01). (OPC 10000-20). OPC-Foundation.

OPC-Foundation. (n.d. a). *Field Level Communications (FLC) Initiative*. Retrieved 3.5.2022 from https://opcfoundation.org/flc/

OPC-Foundation. (n.d. b). *UA Companion Specifications.* Retrieved 16.6.2022 from https://opcfoundation.org/about/opc-technologies/opc-ua/ua-companion-specifications/

open62541. (2021a). *List of Open Source OPC UA Implementations.* Retrieved 23.2.2022 from https://github.com/open62541/open62541/wiki/List-of-Open-Source-OPC-UA-Implementations

open62541. (2021b). *open62541 Documentation, Release 1.2.0-5-gcc3a3d396-dirty.* Retrieved 23.5.2022 from http://www.open62541.org/documentation/

open62541. (2022). *[Draft] FileType implementation #4436* [Pull request]. Github. Retrieved 11.5.2022 from https://github.com/open62541/open62541/pull/4436

Pfrommer, J., Ebner, A., Ravikumar, S. & Karunakar, B. (2018). Open Source OPC UA Pub-Sub over TSN for Realtime Industrial Communication*. 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA), Turin, Italy, 4-7.9.2018.* DOI:10.1109/ETFA.2018.8502479

Profinet. (n.d.). Time Sensitive Networking (TSN). Retrieved 12.5. 2022 from https://us.profinet.com/digital/tsn/

ProSoft. (2022). OPC UA Gateway to the IIOT. ProSoft Technology. Retreived 28.3.2022 from https://www.prosoft-technology.com/Landing-Pages/OPC-UA

Rinaldi, J. (2019). *What Exactly is UA TCP?* Real Time Automation. Retrieved 27.2.2022 from https://www.rtautomation.com/rtas-blog/what-exactly-is-ua-tcp/

Seilonen, I., Tuovinen, T., Elovaara, J., Tuomi, I. & Oksanen, T. (2016). Aggregating OPC UA servers for monitoring manufacturing systems and mobile work machines. *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), Berlin, Germany, 6-9.9.2016*. DOI:10.1109/ETFA.2016.7733739

TSN task group (n.d.) Time sensitive networking (TSN) task group. IEEE. Retrieved 30.5.2022 from https://1.ieee802.org/tsn/

Unified Automation. (n.d.). *OPC Unified Architecture Overview*. Retrieved 23.2.2022 from https://documentation.unified-automation.com/uasdkcpp/1.7.3/html/L2OpcUaFundamentalsOverview.html

Wärtsilä. (2017). *WHSR PROTOCOL TECHNICAL SPECIFICATION* [Restricted Availability]. https://wmap-webhelp.wecs.wartsila.com/content/wmap/whsr_protocol/topics/whsr_protocol_overview.html.

Wärtsilä. (2019a). *HSR Technology* [Restricted Availability]. https://confluence.devops.wartsila.com/x/2l_hAQ

Wärtsilä. (2019b). *WHSR protocol stack* [Restricted Availability]. https://confluence.devops.wartsila.com/x/Jl_hAQ

Wärtsilä. (2019c). *WMAP-CAL* [Restricted Availability]. https://confluence.devops.wartsila.com/x/H12hAQ

Wärtsilä. (2019d). EDL service handler (EDL-SH) [Restricted Availability]. https://confluence.devops.wartsila.com/x/SgT9AQ

Wärtsilä. (2019e). SDK introduction & integration [Restricted Availability]. https://confluence.devops.wartsila.com/x/TEyhAQ

Wärtsilä. (2020a). Synchronous Parameter Access (SPA) [Restricted Availability]. https://confluence.devops.wartsila.com/x/cGIqAw

Wärtsilä. (2020b). *Remote Measurement Service (RMS)* [Restricted Availability]. https://confluence.devops.wartsila.com/x/amlqAw

Wärtsilä. (2021). *Module Status Service Handler (MSS)* [Restricted Availability]. https://confluence.devops.wartsila.com/x/LCP9AQ

Zunino, C., Cena, G., Scanzio, S. & Valenzano, A. (2021). Experimental Characterization of Asynchronous Notification Latency for Subscriptions in OPC UA. *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Vasteras, Sweden, 7-10.9.2021. DOI: 10.1109/ETFA45728.2021.9613502*