

Mapping Modbus to OPC Unified Architecture

Jesper Tunkkari

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 2.3.2018

Thesis supervisor:

Prof. Valeriy Vyatkin

Thesis advisors:

M.Sc. Jouni Aro

D.Sc. Ilkka Seilonen

Author: Jesper Tunkkari

Title: Mapping Modbus to OPC Unified Architecture

Date: 2.3.2018

Language: English

Number of pages: 8+68

Department of Electrical Engineering and Automation

Professorship: Control, Robotics and Autonomous Systems

Supervisor: Prof. Valeriy Vyatkin

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

The importance of data collection and exchange in industrial automation has grown over the years, bringing new requirements to data collection and storage. Modbus is a widely used protocol in the automation industry, but it is insecure and open to attacks. To meet modern day requirements, it needs to be secured. The purpose of this thesis is to create an application that can transfer Modbus data over OPC Unified Architecture, which provides secure communication by default.

To get a good base for the requirements and design, currently available similar applications are evaluated. The main part of the thesis studies how Modbus data should be structured in an OPC UA server application by using Information Modelling. First, a diagram of the structure is created. Then, by using a modelling tool, that diagram is turned into an OPC UA Information Model, defining the base data structure of the application. In the practical part of this thesis, the application is created based on the OPC UA Simulation Server, which is modified to enable configuration of Modbus data transfer. Its GUI is implemented to minimize necessary user interactions and to prevent possible configuration mistakes.

The results show that creating a good Information Model is possible, but is quite time consuming. The finished application shows that OPC UA is very suitable for transferring Modbus data, especially as it can provide metadata in addition to the raw data. It also shows that an OPC UA application can be easy to use if the GUI is designed well enough. As OPC UA showed to be suitable for transferring Modbus data, improvements will most likely be made to the OPC UA Modbus Server in the future.

Keywords: OPC UA, Modbus, Java, information modelling

Tekijä: Jesper Tunkkari		
Työn nimi: Modbusin muuntamista OPC Unified Architectureksi		
Päivämäärä: 2.3.2018	Kieli: Englanti	Sivumäärä: 8+68
Sähkötekniikan ja Automaation Laitos		
Professuuri: Sääntötekniikka, Robotiikka ja Autonomiset Järjestelmät		
Työn valvoja: Prof. Valeriy Vyatkin		
Työn ohjaajat: DI Jouni Aro, TkT Ilkka Seilonen		
<p>Tiedonkeruun ja -vaihdon tärkeys teollisuusautomaatiossa on kasvanut vuosien varrella, tuoden uusia vaatimuksia tiedonkeruulle ja tallentamiselle. Modbus on erittäin laajasti käytetty protokolla teollisuudessa, mutta se on suojaamaton ja avoin hyökkäyksille. Nykyaikaisten vaatimusten täyttämiseksi sitä pitää suojata tietoturvahyökkäyksiltä. Tämän diplomityön tarkoituksena on luoda sovellus, joka pystyy siirtämään Modbus-dataa OPC Unified Architecturen yli. OPC UA:ssa on sisäänrakennettu tietoturva.</p> <p>Tällä hetkellä olemassa olevia, samankaltaisia sovelluksia arvioidaan ja käytetään pohjana suunnittelussa. Työssä keskitytään tutkimaan, miten Modbus-tietorakenteita tulisi mallintaa OPC UA -serverisovelluksessa tietomallintamisen avulla. Ensin tietorakenne määritellään kaavion avulla. Sen jälkeen mallinnussovellusta käyttämällä kaaviosta luodaan OPC UA -tietomalli. Tämä tietomalli määrittää sovelluksen perustietorakenteen. Diplomityön käytännöllisen osan sovellus pohjautuu muokattuun OPC UA Simulation Serveriin, johon on lisätty Modbus-konfiguraatiomahdollisuus. Käyttäjän konfiguraatiovirheet on pyritty estämään minimoimalla toimenpiteiden määrää.</p> <p>Tuloksista selviää, että hyvän tietomallin luominen on mahdollista, vaikka onkin aikaa vievää. Kehitetty sovellus näyttää, että OPC UA soveltuu erittäin hyvin Modbus-datan siirtämiseen, varsinkin koska se pystyy tarjoamaan raakadatan lisäksi myös metadataa. Lisäksi selviää, että hyvällä käyttöliittymäsuunnittelulla OPC UA -sovelluksesta voi tulla helppokäyttöinen. Koska OPC UA näyttää soveltuvan hyvin Modbus-datan siirtämiseen, OPC UA Modbus Serveriä tullaan todennäköisesti kehittämään myös tulevaisuudessa.</p>		
Avainsanat: OPC UA, Modbus, Java, tietomallintaminen		

Författare: Jesper Tunkkari		
Titel: Konvertering av Modbus till OPC Unified Architecture		
Datum: 2.3.2018	Språk: Engelska	Sidantal: 8+68
Institutionen för Elektroteknik och Automation		
Professur: Reglerteknik, Robotik och Autonomiska System		
Övervakare: Prof. Valeriy Vyatkin		
Handledare: DI Jouni Aro, TkD Ilkka Seilonen		
<p>Vikten av datainsamling och -utbyte i industriell automation har växt genom åren, vilket har fört fram nya krav på datainsamling och -lagring. Modbus är ett mycket använt protokoll i automationsindustrin, men det är oskyddat och öppen för attacker. För att möta moderna krav, måste det skyddas från säkerhetsattacker. Syftet med det här diplomarbetet är att skapa en applikation som kan överföra Modbus-data över OPC Unified Architecture, som erbjuder säker överföring av data som standard.</p> <p>För tillfället tillgängliga, motsvarande applikationer evalueras och används som bas för designen. Största delen av arbetet utreder hur data borde struktureras i OPC UA serverapplikationer med hjälp av Informationsmodellering. Först definieras datastrukturen i form av ett diagram. Med hjälp av ett modelleringsprogram omvandlas sedan diagrammet till en OPC UA Informationsmodell, som definierar applikationens basdatastruktur. I den praktiska delen baseras applikationen på en modifierad version av OPC UA Simulation Server, som utökats med Modbuskonfigurationsmöjlighet. För att förebygga användares konfigurationsmisstag utvecklas användargränssnittet så att nödvändiga konfigurationssteg minimeras.</p> <p>Resultaten visar att det är möjligt att framställa av en bra Informationsmodell, även om det är mycket tidskrävande. Den utvecklade applikationen visar att OPC UA är mycket lämpligt för överföring av Modbus-data. Speciellt eftersom den utöver rådata också kan förse metadata. Resultaten visar även att OPC UA applikationer kan vara enkla att använda ifall användargränssnittet designas väl. Eftersom OPC UA lämpar sig för överföring av Modbus-data så väl, kommer OPC UA Modbus Servern högst troligen vidareutvecklas också i framtiden.</p>		
Nyckelord: OPC UA, Modbus, Java, informationsmodellering		

Preface

The opportunity to write my thesis for Prosys OPC is greatly appreciated and I would especially like to thank my instructor Jouni Aro. His guidance and patience with this long lasting project has been invaluable. There were times when a finished thesis seemed too hard to accomplish, but seeing other impossible projects getting finished, like Helsinki's West Metro, gave me the strength to keep going.

I would also like to thank my supervisor Valeriy Vyatkin and second instructor Ilkka Seilonen for their inputs on the matter at hand.

Otaniemi, 2.3.2018

Jesper Tunkkari

Contents

Abstract	ii
Abstract (in Finnish)	iii
Abstract (in Swedish)	iv
Preface	v
Contents	vi
Abbreviations	viii
1 Introduction	1
1.1 Objectives and scope	2
1.2 Research methods	3
1.3 Structure of work	3
2 Background	4
2.1 Modbus	4
2.1.1 History	4
2.1.2 General	4
2.1.3 Data structure	5
2.1.4 Serial	8
2.1.5 Ethernet	8
2.1.6 Other implementations	9
2.2 OPC Unified Architecture	9
2.2.1 History	9
2.2.2 UA Address Space	11
2.2.3 Information Modelling	12
2.2.4 Information Model Extensions	17
2.2.5 Security	18
2.2.6 Services	19
3 Requirements	21
3.1 Use cases	22
3.1.1 Reading production data to MES	22
3.1.2 Providing PLC with process runtime parameters	23
3.2 Evaluating OPC UA enabled Modbus applications	23
3.2.1 KEPServerEX	24
3.2.2 Cogent DataHub Modbus OPC Server	28
3.2.3 Softing dataFEED OPC Suite	31
3.2.4 CommServer OPC UA Server for Modbus IP	34
3.2.5 Summary	36

4	Design	37
4.1	Design decisions	37
4.2	Modbus Information Model Design	38
4.2.1	ModbusDeviceTypes	41
4.2.2	ModbusIoBlockTypes	43
4.2.3	ModbusRegisterConfigurationType	43
4.2.4	HasModbusRegisterConfiguration	43
5	Implementation	46
5.1	Creating the Information Model	46
5.2	Modbus device configuration	49
5.3	Creating the OPC UA Modbus Server	50
5.4	Application deployment	50
5.5	User interface	51
6	Conclusion	56
6.1	Future work	57
	References	58
A	Modbus Information Model	62
B	Modbus NodeSet	64

Abbreviations

ADU	Application Data Unit
AE	Alarms and Events
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
COM	Component Object Model (Microsoft)
DA	Data Access
DCOM	Distributed COM
CRC	Cyclic Redundancy Check
ERP	Enterprise Resource Planning
GUI	Graphical User Interface
HDA	Historical Data Access
HA	Historical Access
IANA	Internet Assigned Numbers Authority
IO	Input, Output
IP	Internet Protocol
LAN	Local Area Network
MBAP	Modbus Application Protocol
MES	Manufacturing Execution System
MOM	Manufacturing Operations Management
MSB	Most Significant Bit
ODBC	Open Database Connectivity
OPC	Open Platform Communication or OLE for Process Control
OPC UA	OPC Unified Architecture
PDP	Process Data Protocol
PDU	Protocol Data Unit
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition
SCP	Secure Copy
SDK	Software Development Kit
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UML	Unified Modelling Language
VPN	Virtual Private Network
WAN	Wide Area Network
XML	Extensible Markup Language

1 Introduction

Data collection and exchange have always been a part of industrial automation, but in recent years they have grown to be key aspects of a successful business. There are several reasons for companies to collect data from their manufacturing processes. Such reasons include, for example, obligations due to regulations in specific industries such as the food industry. In the automotive industry, collected data can be used for anything ranging from material management to manufacturing process development. Whatever the reason, data needs to be transferred, to a different extent, across all the layers of the automation pyramid (seen in figure 1).

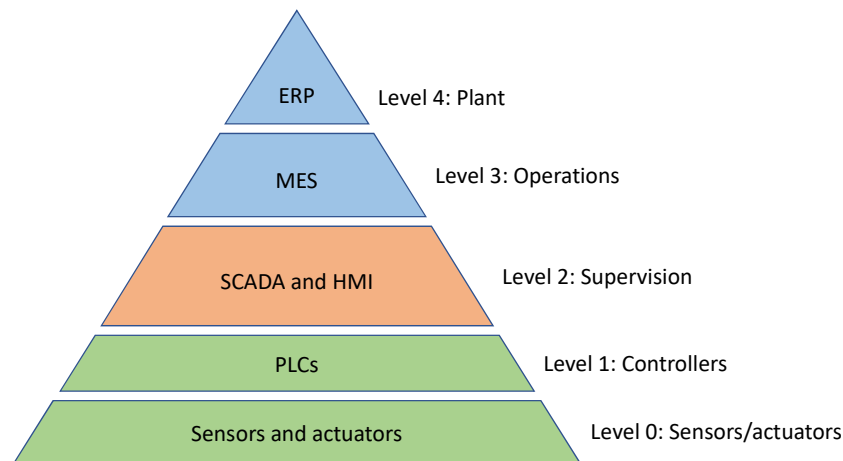


Figure 1: Levels of the automation pyramid.

As automation technology advances, more and more devices are connected to the corporate and industrial networks every day. Thus there is a growing importance of managed networks and secure transfer of data. The automation protocols over Fieldbus or Industrial Ethernet, such as Modbus, PROFIBUS, PROFINET or EtherCAT were designed for speed and reliability between actuators, controllers and SCADA, and for delivering the data with strict deadlines. They were not designed to be integrated into regular LAN networks but for separated automation networks. As the networks are isolated, there has been no need to apply security or encryption, allowing minimized overhead in data transfers. The most demanding applications using Industrial Ethernet might use special network switches that implement PDP (process data protocol) on the transport and network layer instead of TCP/IP and may even have special real-time Ethernet controllers on the physical and link layers for as fast transfer times between the devices as possible [1]. Integrating such a network into an office network, would increase the traffic radically and slow down the whole system.

The security aspects have historically not been of major concern to the industrial automation industry. Only after the report of Stuxnet in 2010 security technologies and management became a significant factor in the industry [2]. As security breaches

can cause significant profit loss, material damage or, in worst case, even deaths, it is clear that this matter needs to be taken seriously.

Today the task of providing data to the corporation management systems like MES and ERP is not as simple any more as confidentiality, integrity and availability need to be considered. The first and most important security measure is to keep the automation and office networks separated. To accomplish this gateways are used. Gateways have traditionally been used for protocol conversion, but today they are also used to provide a layer of security [3]. An enterprise may be distributed all over the world and their factories can be located in different cities than their operational divisions. This means that when data needs to be exchanged between them it has to be transferred securely. There are several ways to accomplish secure transfer of data, for example by using virtual private networks (VPN), transport layer security (TLS) or Secure Copy (SCP), although none of them are really intended for transferring live automation data.

The industrial protocol Modbus is a very widely used protocol because it is simple, royalty-free and vendor independent. But, even though it is widely used, it has a big disadvantage in that it lacks any kind of security attributes [4]. To transfer process data from an industrial control system using Modbus, to the upper levels in the automation pyramid, a more suitable method is thus necessary. In such a case, a very good solution is to use OPC UA, which is developed to provide a secure and agile way of communication. OPC UA supports creation of Information Models which not only provides raw process data, but can also provide semantics of the application.

By mapping Modbus to OPC UA, the simple and widely used protocol can be made easily available over a secure channel for all the upper levels in the automation pyramid. In addition to the data provided by the Modbus devices, the application can provide metadata about the process as well. As there is such a large amount of Modbus devices available on the market and they are used in an endless amount of different ways, it is necessary for the user to be able to configure the mapping of Modbus data to OPC UA. This way the users can select the necessary data and customize it according to their need.

1.1 Objectives and scope

The purpose of this thesis is to develop an application for mapping an open communication protocol to OPC UA. The idea is to enable users to easily provide a secure means for transferring data to different types of applications ranging from SCADA to ERP systems. To limit the extent of this thesis a scope is set. The most important part of the scope is to specify the open communication protocol to be mapped, the most common automation protocol Modbus will be used. Specifically only the TCP/IP based implementations of Modbus are included. In the aspects of OPC UA, only the part that is providing data is included. Specifically only the creation of an OPC UA server with Data Access variables. The implementation of an OPC UA client application for handling or reading Modbus data is not covered at all. Neither all existing data type implementations will be provided. The supported data types will be decided later on.

As this application will use OPC UA, large weight will be put on designing the application in a way that fits the OPC UA architecture. The thesis will study these three main points:

1. How should data be structured and modelled in an OPC UA server application or more specifically how an Information Model is designed with the right level of abstraction.
2. Can the usage and configuration of such an application be developed in a way that it is easy to set up.
3. Furthermore, the benefits of OPC UA over simply wrapping Modbus with a secure protocol, such as HTTPS, will be evaluated.

1.2 Research methods

For the most part of this thesis, information is acquired by literature review and the specifications of both Modbus and OPC UA. For the design part of the thesis and for decision making, fellow co-workers at Prosys OPC Ltd. are consulted. The design process follows an agile development method where the design is reviewed and improved iteratively. In projects of this size, agile is a good choice as each design iteration can be reviewed quickly and the design is then easier to improve.

Product reviews of currently available similar applications are conducted as an important part of the thesis. The reviews are used to create a good starting point of the design goals of the application. They give valuable information on usability, which functionalities are necessary and if there are some functionalities that could be implemented that these are lacking.

The mapping between Modbus and OPC UA is designed by creating an Information Model. As the creation of Information Models is very common in OPC UA, a graphical notation specific to OPC UA has been created to visualize the design. This notation is used to help designing the Information Model and an application called UaModeler is then used to create the Information Model.

1.3 Structure of work

This thesis is divided into six main chapters. The first chapter introduces the reader to the subject and gives a short answer to what has been created and why it has been developed. The second chapter gives the reader some necessary background knowledge of both Modbus and OPC UA. The third chapter talks about the requirements, in which cases it could be used and what kind of similar applications already exists. Chapter four contains the design process. It describes design decisions that aim for making the application preferable over similar applications on the market and it also describes the design of the Information Model. In chapter five the implementation phases from model generation to the finished application is presented. The Last chapter wraps up the work by discussing the outcome of the project.

2 Background

To offer an insight on how Modbus and OPC UA differ from each other and how they could be mapped together, some background information is necessary. As their specifications are very long and detailed, only a short and informative version of them is presented. The background aims for giving the reader enough information about the subject to be able to understand the design and implementation processes.

2.1 Modbus

2.1.1 History

The Modbus protocol is an old industrial standard protocol that is developed by Modicon and was introduced in 1979 [5]. Modicon that today is a part of the widely known Schneider Electric, is also one of the founders of the member-based, non-profit organization Modbus Organization, that took over the development and maintenance of the Modbus protocol [6]. The advantage of using Modbus is that it's not industry specific and it uses a messaging structure common to all devices. It can be implemented over both serial and Ethernet communication and its simple and adaptable implementation has made it very widely used in the automation industry.

2.1.2 General

The Modbus protocol uses a master/slave technique (also called client/server) where the slave is a passive device that provides data upon request while the master is the active part that initiates the data transactions. As seen in figure 2, the master sends a request to a slave, to which the slave then creates either a normal response or an exception response.

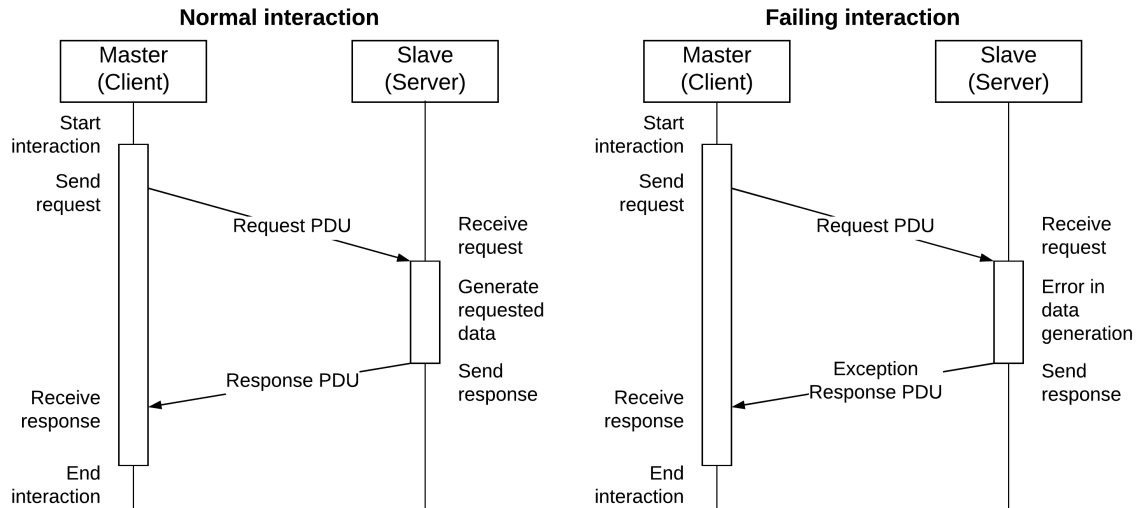


Figure 2: Master - Slave data interactions.

Modbus was initially designed to transfer data over the serial interfaces EIA-232, EIA-422 or EIA-485 but later on an Ethernet based implementation called Modbus

TCP/IP has been developed as well [7]. That is why the Modbus protocol usually is divided in two parts, serial and Ethernet based.

2.1.3 Data structure

The Modbus protocol implements a simple protocol data unit (PDU) independent of the underlying communication layers. The Modbus protocol defines three different types of PDUs:

- Modbus request
- Modbus response
- Modbus exception response

In each of these cases the PDU is divided into two fields, *function code* and *data*. The function code field in the PDU is one byte long with valid values between 1 and 255, meaning 0 is not a valid function code [8]. A function code is set in the request and if no exception occurs, the same function code is returned in the response. As visualized in table 1, if an exception occurs, it is distinguished by the MSB of the function code field in the request being set to one in the exception response.

Table 1: Function code of a Modbus transaction example.

	Function code (hex)	
Request	0000 0110	(0x06)
Response	0000 0110	(0x06)
Exception response	1000 0110	(0x86)

As shown in figure 3, the PDU is wrapped in an application data unit (ADU) that introduces certain additional fields depending on the underlying protocol [8]. These additional fields can be generalised to *Additional address* and *Error check*. Depending on if serial or Ethernet communication is used different fields are implemented in the ADU, these differences are discussed in the sections 2.1.4 and 2.1.5 explaining the different communication layers.

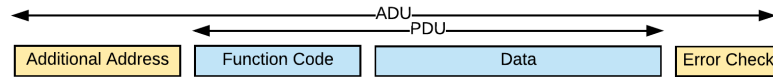


Figure 3: Application data unit.

Modbus defines four types of data with varying features (table 2). The inputs, both discrete inputs and register inputs represent physical inputs on the Modbus device. The coil outputs and holding registers can be either internal (virtual) or physical outputs. Let's take an electrical drive as an example. The physical coil outputs are used e.g. for starting, stopping or locking the drive while discrete inputs

provide the run status. The holding registers are used to set analog values, e.g. the speed setpoint of the drive while input registers provide data like real-time speed value.

Table 2: Modbus data types.

Type	Access	Size
Discrete input	read-only	1-bit
Coil output	read-write	1-bit
Input registers	read-only	16-bit
Holding registers	read-write	16-bit

Each data type is allocated to a block of memory addresses. Illustrated in figure 4, the blocks use a 16-bit address that restricts each type of data to 65536 data items. So e.g. a 16-bit holding register can have an address between 0–65535. Now to make things a bit confusing, there are a few ways of addressing the Modbus data items. Two of them can be seen in table 3.

Address (16-bit)	Data
Input Discretes (1-bit)	
0x0000 (0)	0b0
...	...
0xFFFF (65535)	0b0
Coils (1-bit)	
0x0000 (0)	0b0
...	...
0xFFFF (65535)	0b0
Input Registers	
0x0000 (0)	0x0000
...	...
0xFFFF (65535)	0x0000
Holding Registers	
0x0000 (0)	0x0000
...	...
0xFFFF (65535)	0x0000

Figure 4: Example of Modbus data blocks.

The *old addressing* has its advantage in that the address is meant to give information on both the data type and the address by letting the leading number specify the data type and the remaining four numbers specify the address. One disadvantage is that the first item in the data block, e.g. holding register 0 is addressed to as 40001 which might get confusing or misleading [7]. Another disadvantage is that this way of addressing restricts the data items to 9999 even though 65536 addresses should be available for each data type. To fix the addressing issue, it can be extended from 40001 to 400001 by adding a zero in between. But as it this still seems a

bit confusing that a zero could be added just like that, there is a way to make it a bit more unambiguous. The *Fixed addressing* in table 3 is an unofficial way of bypassing the confusion between normal and extended addressing. It adds an x after the specified data type to separate the data type and the register address. Currently the best way of addressing is to specify the data type and the address separately.

Table 3: Modbus register addressing.

Type	Old addressing	Fixed addressing
Discrete input	00001–09999	0x0001–0x65536
Coil output	10001–19999	1x0001–1x65536
Input registers	30001–39999	3x0001–3x65536
Holding registers	40001–49999	4x0001–4x65536

To specify what type of request is sent the protocol implements a set of function codes. Table 4 shows the most commonly used and implemented function codes. The function codes between 1 and 127 are either public function codes that are publicly documented, unique and maintained by the Modbus Organisation or then they are user defined [8]. The value ranges 65–72 and 100–110 are reserved for the user defined function codes and the rest are public function codes. The values from 128 up are reserved for the exception responses.

Table 4: Common Modbus function codes (FC).

	Function	FC
Physical discrete inputs	Read Discrete inputs	02
Internal bits or physical coils	Read Coils	01
	Write Single Coil	05
	Write Multiple Coils	15
Physical input registers	Read Input Register	04
Internal Registers or physical output registers	Read Holding Registers	03
	Write Single Register	06
	Write Multiple Registers	16

The data field (figure 3) in each type of PDU, depending on the function code, contains either data or information necessary for the slave to be able to provide the correct data to the master. The PDU data field of a Modbus Request contains a register starting address and the quantity of bytes to be read. The data field in the corresponding Modbus response contains the amount of bytes and the data itself.

2.1.4 Serial

The Serial line Modbus specifications can be divided into RTU and ASCII modes. Modbus RTU is the most lightweight as it transfers binary data as opposed to Modbus ASCII that uses ASCII characters. Because it is binary it might sometimes be referred to as Modbus-B as well. The ASCII protocol usually has to transfer data that is double the length of what the RTU data would be [7]. All Modbus serial devices must implement at least the RTU mode [9]. The serial Modbus protocols use an additional address field and error check in the ADU.

The ADU in figure 3 illustrates an ADU that is used in the serial protocol so it implements an additional address field and an error check field in addition to the PDU. The additional address field is a 1-byte address for slave devices so that each of them can be uniquely identified over a serial interface such as EIA-485. This can be seen in figure 5, only here the additional field is called unit id. Valid addresses of slave devices are 1–247 and the rest are reserved.

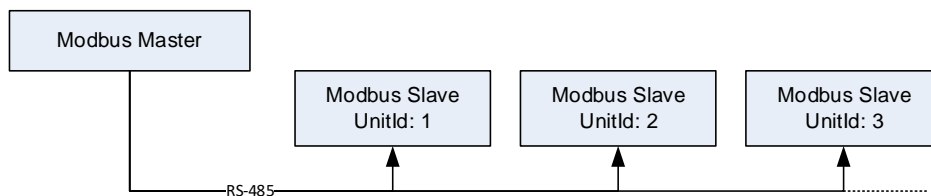


Figure 5: Modbus devices in EIA-485 serial line.

The Modbus serial modes implement two standard error detection features. The first one is an error check of either a 2-byte CRC used in RTU mode or a 2-char LRC used in ASCII mode. Parity checking is used as the other standard error detection feature. The parity checking adds a parity bit to each character transmission. There are three different types of parity checking, *even*, *odd* and *none*, where the only compulsory option to be implemented is *even*. [9]

2.1.5 Ethernet

The Modbus protocol defines two standard Ethernet protocol versions:

- Modbus TCP/IP, also called Modbus TCP
- Modbus over TCP/IP, also called Modbus RTU over TCP or Modbus RTU/IP

First of all the Modbus TCP ADU package differs from the serial ADU in the sense that the error check is removed as there already is a 32-bit CRC implemented on the Ethernet layer. The second difference is that the additional address field is replaced by a so called MBAP header, which is 7 bytes long and is significantly longer than the 1 byte long address in the Modbus serial PDU [10]. The difference between Modbus TCP and Modbus over TCP is that Modbus over TCP wraps the serial ADU in a TCP package and not the standard Ethernet ADU.

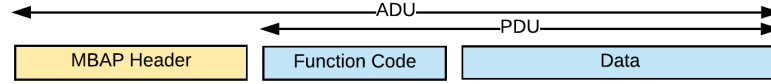


Figure 6: Modbus TCP ADU.

The MBAP header consists of the four fields listed in table 5 [10]. The transaction identifier is used to keep track of which request belongs to which response. The protocol identifier is used for intra-system multiplexing and the Modbus protocol is identified by 0. The length gives the quantity of bytes that follow, this includes both unit identifier and data fields. The unit identifier is used in Modbus over TCP for identifying serial Modbus slaves behind an Ethernet \leftrightarrow Serial Modbus gateway.

Table 5: MBAP Header fields.

Fields	Length	Description	Client	Server
Transaction Identifier	2 Bytes	Identification of a Modbus request/response transaction	Initializes <i>Req</i>	Recopied <i>Req</i> \rightarrow <i>Res</i>
Protocol Id	2 Bytes	0 = Modbus protocol	Initializes <i>Req</i>	Recopied <i>Req</i> \rightarrow <i>Res</i>
Length	2 Bytes	Number of following bytes	Initializes <i>Req</i>	Initializes <i>Res</i>
Unit Id	1 Byte	Identification of remote slave connected through serial	Initializes <i>Req</i>	Recopied <i>Req</i> \rightarrow <i>Res</i>

An important note is that all Modbus TCP traffic is sent to port 502 which is registered at IANA as a System port [10][11]. In practice this means that Modbus slaves should always be listening on port 502.

2.1.6 Other implementations

Modbus also implements other protocol versions that are out of the scope of this thesis. Some of the protocols worth mentioning though are Modbus+ and Enron Modbus [12][13].

2.2 OPC Unified Architecture

2.2.1 History

The IEC standard 62541 OPC Unified Architecture was first released in 2008 and is a platform independent service-oriented architecture aimed to replace its predecessor OPC Classic [14]. OPC UA integrates all the functionalities of OPC Classic into a single, extensible framework. The reasons for the development of OPC UA is that OPC Classic has its restrictions. The main restriction is that OPC classic is bound to Windows based computers as it uses Microsoft's COM/DCOM technology. It also implements several different standards of which the most common ones are OPC DA (Data Access), OPC HDA (Historical Data Access) and OPC AE (Alarms and

Events). OPC Classic was initially designed for communication on the automation level where as OPC UA is designed to enable usage throughout all the layers of the automation pyramid (figure 1).

Since OPC UA was released it has had a steady growth and recently it has finally found its place in the industry as the de facto standard for industrial communication by replacing its predecessor [15]. This was verified in June 2017 when OPC UA was declared to be the means of communication for the German Industrie 4.0 as the Reference Architecture Model for Industry 4.0 (RAMI 4.0) only recommends OPC UA for implementing the communication layer [16]. Furthermore many of the big corporations have shown interest in OPC UA and are actively participating in the development. One of the more active participants is Microsoft, that has made OPC UA available for the .NET Standard and in their cloud platform Azure [17].

The current OPC UA Specification 1.03 was released by the OPC Foundation in 2015. The specification is divided into 13 parts (listed in table 6) describing the functionalities. The most important parts of the specification are part 3 and 4 [18]. These parts define the *Address Space* model and the services. They describe how the data and information hierarchies should be modelled and exposed and how the server and client should interact with each other.

Table 6: OPC UA Specification parts.

Part	Description
<u>Core Specification Parts</u>	
1	Overview
2	Security Model
3	Address Space Model
4	Services
5	Information Model
6	Mappings
7	Profiles
<u>Access Type Specification Parts</u>	
8	Data Access
9	Alarms and Conditions
10	Programs
11	Historical Access
13	Aggregates
12	Discovery
<u>Companion Specifications</u>	
	Devices
	Analyser Devices
	IEC 61131-3
	ISA-95 Common Object Model
	...

2.2.2 UA Address Space

The main task of the OPC UA Address Space is to expose a collection of *Objects* and their data in a standard way on an OPC UA server to the OPC UA clients [19][20]. The Objects can contain *Variables* and *Methods*. The OPC UA Address Space is constructed in an object-oriented way and the Objects are represented as a set of *Nodes*, which are defined by *Attributes* and *References*. Figure 7 illustrates the Node model, here the Attributes describe the Node itself and the References describe its relation to other Nodes.

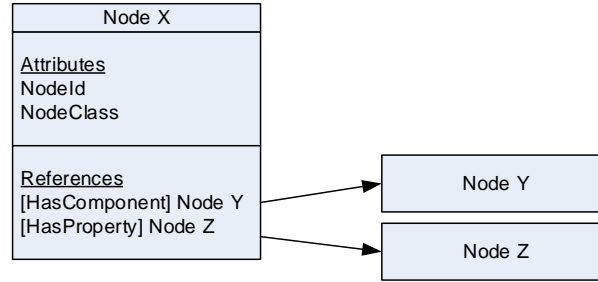


Figure 7: OPC UA Node Model.

All Nodes in the OPC UA Address Space belong to a *NodeClass* and the Node implements all the attributes of the NodeClass it belongs to. The main NodeClasses in OPC UA are Objects, Methods and Variables. The OPC UA specification defines a *Base* NodeClass from which all other NodeClasses are derived, table 7 shows the attributes of the Base NodeClass [21]. The Base NodeClass does not specify any references.

Table 7: BaseNodeClass Attributes.

Name	Data Type	Usage
NodeId	NodeId	Mandatory
NodeClass	NodeClass	Mandatory
BrowseName	QualifiedName	Mandatory
DisplayName	LocalizedText	Mandatory
Description	LocalizedText	Optional
WriteMask	UInt32	Optional
UserWriteMask	UInt32	Optional

In addition to the three main NodeClasses the OPC UA Specification defines five more, these are *ObjectType*, *VariableType*, *ReferenceType*, *DataType* and *View*. To be able to visualize these, the OPC UA Specification (Part 3, Annex D) defines a standard graphical notation for the NodeClasses and references [21]. The NodeClasses, represented by their graphical notations are listed in figure 8 below and a detailed description of them can be found in the OPC UA specification. The Type NodeClasses are also described later on when Information Modelling is introduced in section 2.2.3.

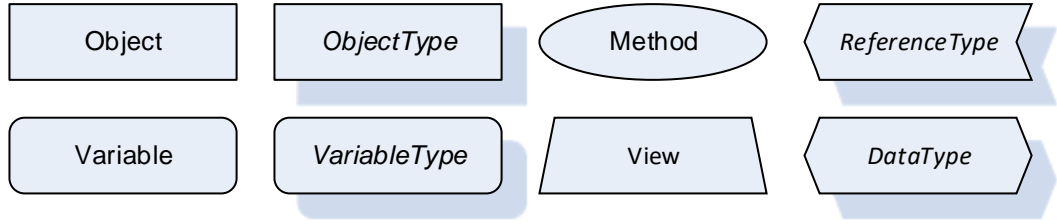


Figure 8: OPC UA Graphical notation of NodeClasses.

The most important attribute of a Node is the *NodeId*. The *NodeId* attribute is a unique identifier that is used to distinguish the Nodes on the server from each other [18]. The *NodeId* consists of three parts: namespace index, enumerated identifier type and identifier. There are four different identifier types which define a specific identifier structure, these are shown in table 8. Although the identifier is supposed to be unique, the same identifier can be used in the context of another namespace [22]. For example the Node with *NodeId* `ns=1;s=WasteWaterFlow` in table 8 would not be the same as `ns=4;s=WasteWaterFlow`.

Table 8: *NodeId* Identifier Types.

Name	Description	<i>NodeId</i> , XML Notation example
Numeric, i	Numeric value	<code>ns=1;i=5001</code>
String, s	Text string	<code>ns=1;s=WasteWaterFlow</code>
GUID, g	Globally unique identifier	<code>ns=2;g=098113a3-3b1a-123d-adf2-724d365cc214</code>
Opaque, b	Namespace specific ByteString	<code>ns=4;b=K/RbAEsRVoePQePfx18ofA==</code>

The OPC UA Address Space Model can be viewed as a meta model that consists of the NodeClasses and their fixed set of attributes. In addition to the NodeClasses, some standard Nodes are defined in the meta model as well [18] [23]. This meta model is usually also referred to as the base *Information Model*. The Nodes defined in the meta model are *ReferenceTypes* and *TypeDefinitions*. OPC UA servers implement this Address Space Model as a base which can be extended. OPC UA Information Models define certain domain-specific types, constraints and well-defined instances. Once an Information Model has been created, the same model can be used in several different OPC UA servers. The *Data* part, illustrated in figure 9, is server specific and consists of the concrete instances that will provide the process values that have been instantiated.

2.2.3 Information Modelling

Information Modelling is a fundamental part of OPC UA. While OPC Classic only provides "raw" data, OPC UA gives the possibility to expose the semantics of the provided data while not enforcing it [18]. The Information Models are created to provide extensive information about the domain for which it is intended and even the most complex multi-level object structures can still be modelled and extended

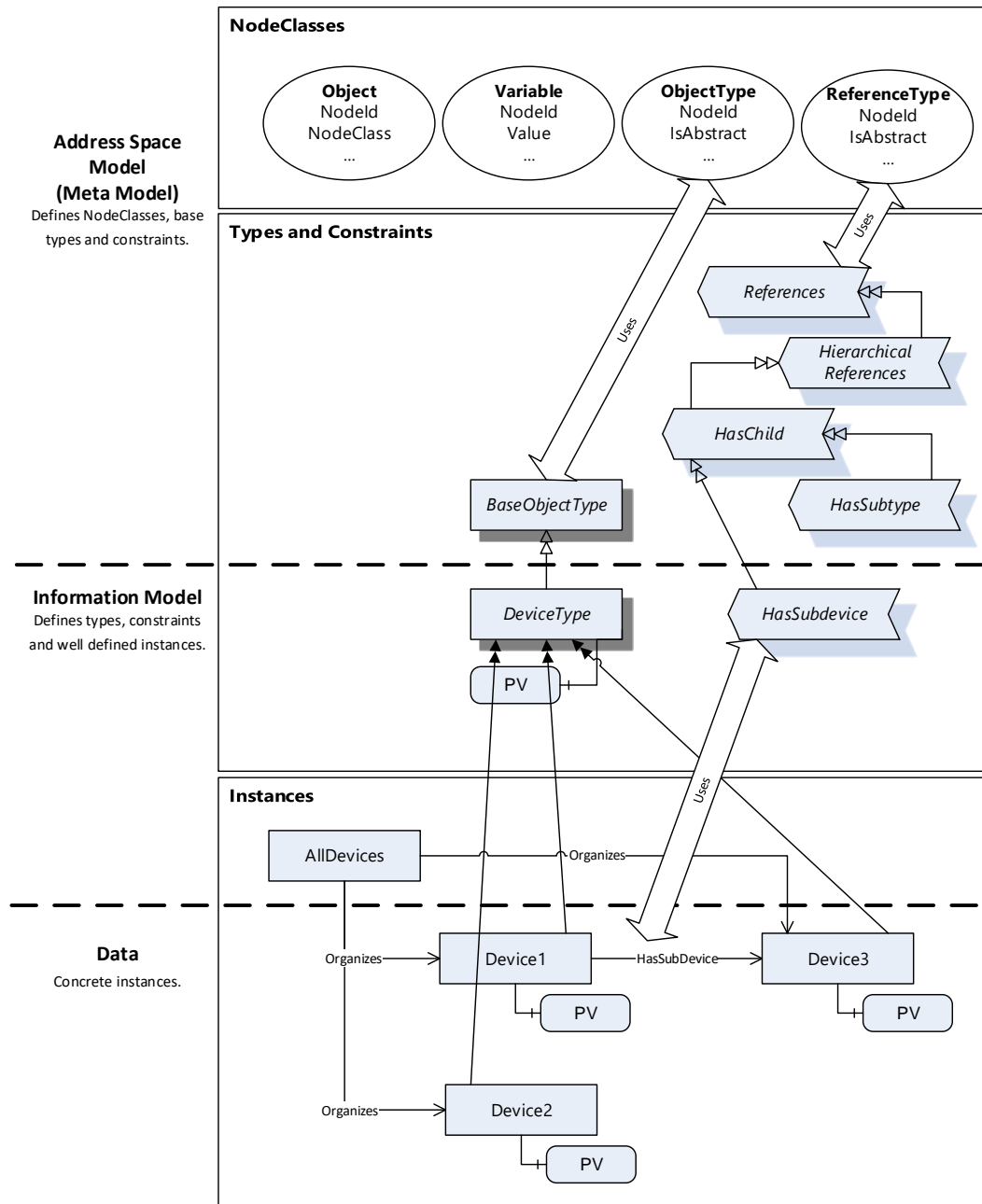


Figure 9: OPC UA Server Address Space.

[14]. What this means is that OPC UA does not only provide type information on the data type level, like Integer or Double for a Variable, but it also provides type information on the object level. It can for example specify that an object is a device that provides temperature and humidity measurements by using a reference to an ObjectType. In OPC UA the NodeClasses ObjectType and VariableType are commonly called TypeDefinitions which as the name suggests, define the type of a Node (Object or Variable).

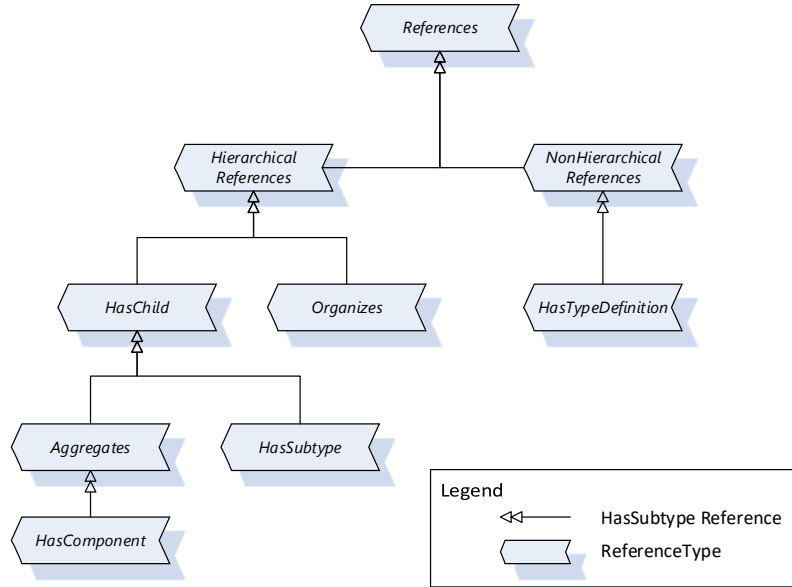


Figure 10: OPC UA Base ReferenceType hierarchy.

As stated earlier, Nodes include attributes and references. To expose the semantics of the references of a Node, OPC UA uses the NodeClass *ReferenceType* [18]. A Reference describes the relation between two nodes and the ReferenceType gives more specifics about that relation. ReferenceTypes are instantiated as Nodes and are visible in the OPC UA Address Space where as a Reference is not [21]. The OPC UA specification describes a set of predefined ReferenceTypes that are organized in a hierarchy. The Base ReferenceType hierarchy is illustrated in figure 10. When using the graphical notation of the OPC UA specification the References are described with arrows of different types, these are illustrated in figure 11. If the OPC UA server needs to define its own ReferenceTypes, the base ReferenceType hierarchy will be extended by adding the necessary references as subtypes.

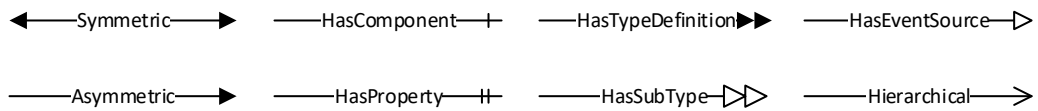


Figure 11: OPC UA Graphical notation of ReferenceTypes.

The ObjectType provides definitions for Objects [21]. It can be seen as an abstraction or generalization of an Object. A very good example of this can be seen in figure 9 where all the Device instances have the TypeDefinition *DeviceType*. The TypeDefinition makes the Device instances inherit the structure of the DeviceType, in this specific case all the created device instances will automatically be given the PV variable.

The ObjectType can be divided into simple and complex ObjectTypes [18]. A simple ObjectType is mainly used for organizing the address space and only defines semantics for the Object that references it. A good example of a simple ObjectType is the *FolderType*. Its purpose is only to organize the Address Space.

An Object references its type by using the a HasTypeDefinition reference. Each Object can only define exactly one HasTypeDefinition reference. Let's take the root structure in figure 12 as an example. the Root node has the HasTypeDefinition reference to FolderType and the Root object use a ReferenceType called *Organizes* to reference nodes that should be organized by it. By looking at the figure it can also be deduced that base Information Model is extended with the OPC UA for Devices Information Model. That part of the specification defines a Device Model in which the *DeviceType* ObjectType can be found [24].

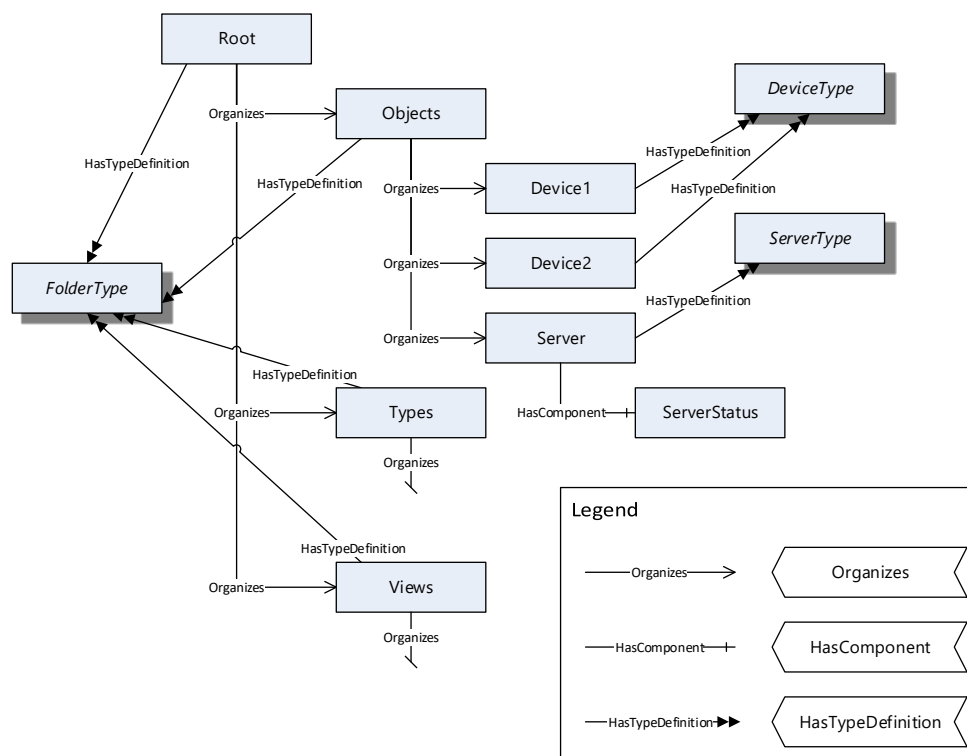


Figure 12: OPC UA Server Root Address Space structure.

The ObjectTypes support inheritance, meaning an ObjectType can be extended to have a more complex or specific structure than its parent. Let's say an ObjectType called *SensorType* is extended to a *TemperatureSensorType*. An example of this can be seen in figure 13, here the temperature sensor type inherits the PV Variable. It does not add any attributes or references but extending it will specify that it is a temperature sensor instead of just any sensor. If it would add an attribute, that could be something that provides either a unit or a range property for giving additional information about the object instance.

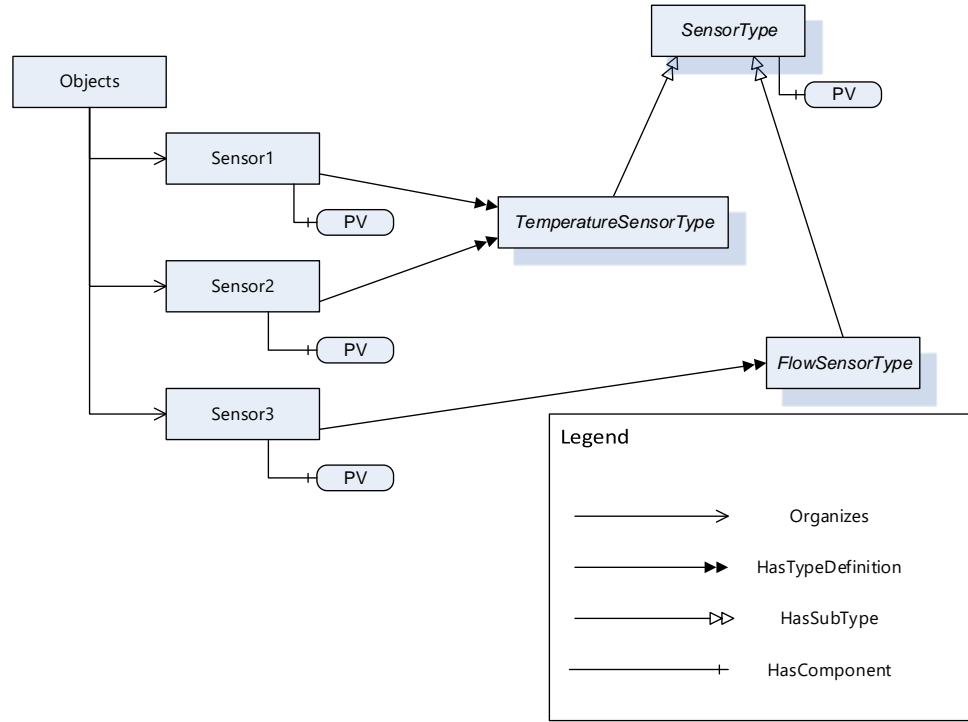


Figure 13: Example of OPC UA type inheritance.

The complex ObjectTypes define a structure of Nodes that is available beneath them on each instance of the ObjectType [18]. Figure 14 illustrates an ElectricalDriveType. The type defines a *State* Variable, a *Parameter* Object and *Start & Stop* Methods. The Parameter Object further defines two variables: *Current limit* and *DC Braking*. When a Drive Object referencing the ElectricalDriveType is created, the whole structure defined by the ObjectType is automatically created by the server, including all Nodes and References.

Like the ObjectType, the VariableType is divided into simple and complex types as well. Just as the complex ObjectType, the complex VariableType defines a structure of Nodes beneath it and the simple VariableType only defines some semantics or restrictions of a Variable.

InstanceDeclarations and *ModellingRules* are an important part of Information Modelling. The InstanceDeclarations are Objects and Variables that are defined in a complex Object- or VariableType. In figure 14 the Nodes beneath the ElectricalDriveType are Object, Methods and Variables and this means they should be Instances and not Types, even though they belong to a TypeDefinition (ObjectType). But as these only are instantiated when a node with the TypeDefinition is created, these are called InstanceDeclarations which basically means that it is declared that these Instances are to be created when an Instance of the ElectricalDriveType is created. A more precise definition of InstanceDeclarations is that they are referenced from the ObjectType by a hierarchical reference in forward direction either directly or indirectly by another InstanceDeclaration [18].

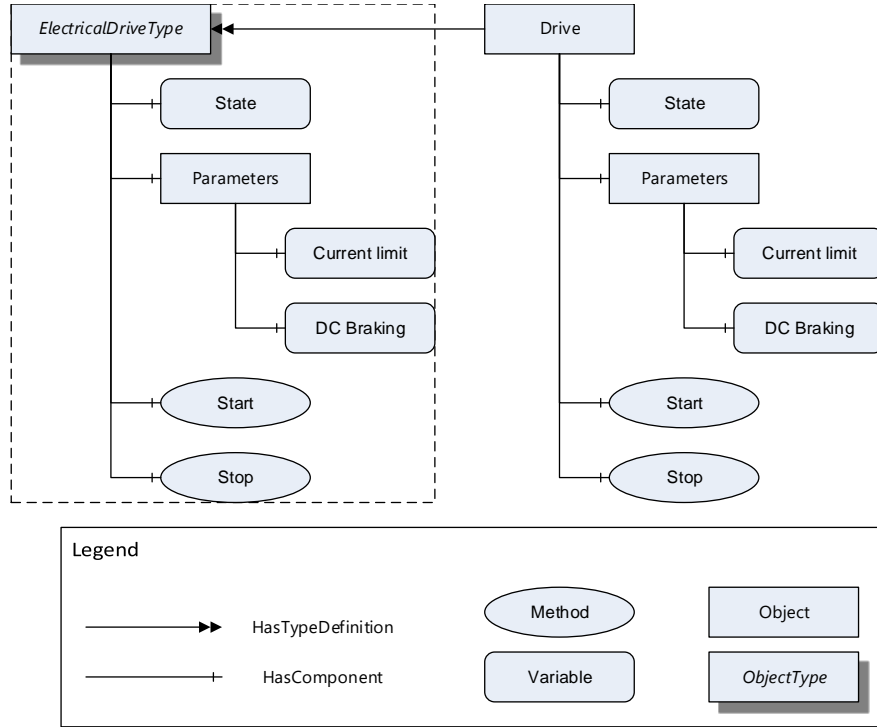


Figure 14: Example TypeDefinition of an electrical drive.

An InstanceDeclaration must always define a ModellingRule. A ModellingRule specifies what happens to the InstanceDeclaration with respect to instances of the ObjectType [18]. The ModellingRule defines a mandatory property called *NamingRule* that describes the modelling rule. The NamingRule allows three values: *Mandatory*, *Optional* and *Constraint*. The Mandatory rule forces an InstanceDeclaration to be created while an Optional InstanceDeclaration can be left out and the Constraint restricts creation of InstanceDeclarations in a specified way.

2.2.4 Information Model Extensions

The OPC UA Specification further defines some standard Information Model extensions. These can be seen in table 6 under the Access Type Specification Parts section. These models define a standard way to represent capabilities and diagnostics of an OPC UA server Address Space and how specific information for these parts are modelled [18].

On top of the standard Information Model extensions there are Companion Specifications (table 6) that define domain specific Information Models. They are developed by other organizations and evaluated by the OPC Foundation. Figure 15 shows the OPC UA architecture and how the OPC UA base, the extension Information Models, companion specific and custom Information Models relate to each other. Out of these specifications the OPC UA for Devices is relevant in the extent of this thesis. The Devices specification defines a base Device Model that should provide a general view of any kind of devices [24].

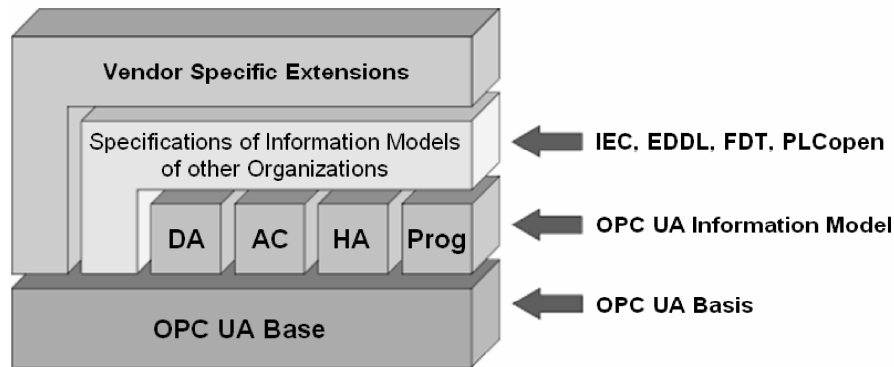


Figure 15: OPC UA Information Model Architecture [18].

2.2.5 Security

The OPC UA Specification defines two transport protocols: HTTPS and UA TCP. OPC UA TCP is a TCP based protocol that creates a full duplex communication channel between an OPC UA client and OPC UA server [25]. Figure 16 shows the OPC UA security architecture. The UA TCP ties functions on the transport layer closely together with the communication layer for optimal performance and as opposed to HTTPS it allows response messages to be sent in any order. Only the UA TCP will be covered in this section as it is usually the preferred transport protocol. It is the fastest and most efficient way to transport data over OPC UA, while binary encoded HTTPS is slightly slower [18].

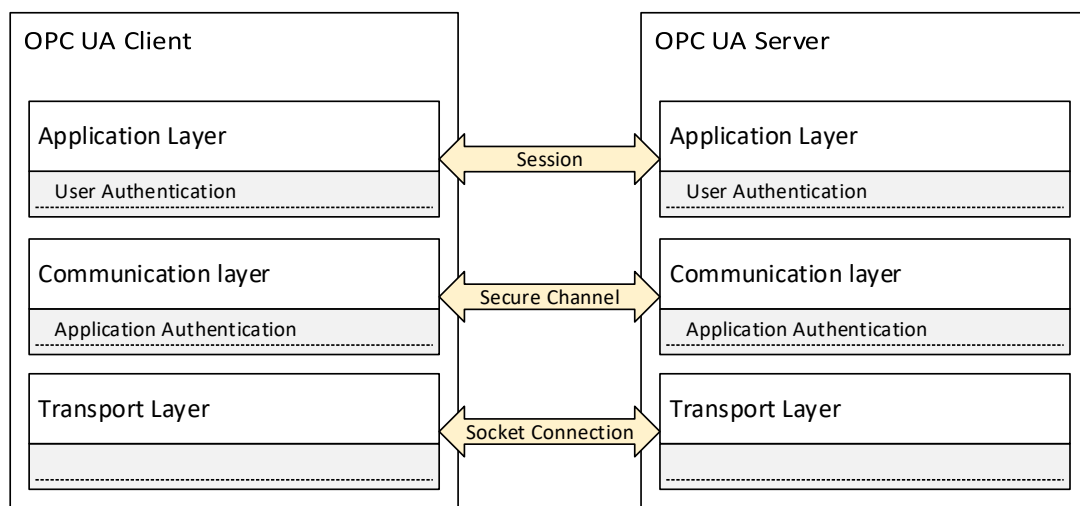


Figure 16: OPC UA security architecture [18].

The information exchange between an OPC UA server and client is done on top of a *Session*, which is created on the *application layer* in the OPC UA Security architecture [20]. This layer also handles the user authentication and user authorization security

tasks. All these functionalities are bundled to a high level API called *UA SDK*, which is built on top of a low level API called *UA Stack*.

The session is created on top of a *Secure Channel* which is created on the *communication layer* by a service in the UA Stack. The UA Stack handles confidentiality, integrity and application authentication. The UA TCP connection is created on a socket connection that is implemented on the *transport layer* and while it handles sending and receiving messages, the UA Stack handles re-establishment of lost connections without interrupting the Secure Channel. [20]

The OPC UA Specifications define three security modes [25]:

- None
- Sign
- Sign & Encrypt

The server application provides some of the security protocols or all of them and the client application then chooses which to use. What security protocol to use is usually decided by the system administrator and varies depending on the environment. As circumstances may change it can always be changed at a later point as well.

The server defines Security Policies which provide the algorithm used for signing and encrypting. Since some of the algorithms that today are considered safe might become unsafe in the future, several algorithms are usually provided. The currently defined security policies are: [26]

- None
- Basic128Rsa15
- Basic256
- Basic256Sha256

2.2.6 Services

OPC UA uses *Services* to offer OPC UA clients a way of accessing data of an OPC UA servers Information Model [18]. The OPC UA services are an abstract definition where the communication interfaces are defined so that it can be implemented regardless of transport protocol, language or environment. They can be considered as the core of OPC UA as without them data could not be exchanged. It can be seen that the service specification is the base layer of the OPC UA communication layer stack in figure 17. The second layer provides the mapping to transport protocols. On the following layers programming language specific stacks are then implemented. Furthermore on top of the stack an SDK is implemented to provide full use of the OPC UA protocol [16].

The OPC UA services are logically organized into so-called service sets where related services are grouped together into a set [27]. The service sets are only used for the standard itself and are not used in the implementations. Table 9 shortly

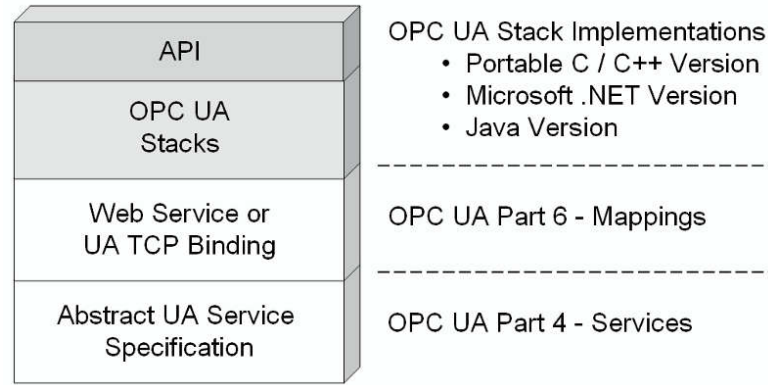


Figure 17: OPC UA communication layering [18].

describes these service sets. If an even broader view of the services is taken they can be divided into two groups: communication establishment and information exchange [18].

Table 9: OPC UA Service Sets [27].

Service Sets	Description
Discovery	Services that can be used to discover endpoints and their security configurations implemented by a server.
Secure Channel	Services used to create a secure communication channel that ensures the confidentiality and integrity of all messages transmitted over the channel.
Session	Services for establishing a session on an application layer connection.
View	Browse services enable clients to navigate through the Address Space (or its subsets like views) of a server.
Attribute	Read/Write data and metadata, HistoryRead and HistoryUpdate services.
Method	Enabling invocation of function calls to methods on the server.
Monitored Item	Services for managing monitored items and how they should be handled
Subscription	Services for managing subscriptions to monitored items and how the publishing of them should behave.
Query	The services can be used to find information in complex Address Spaces like e.g. bulk data access.
Node Management	Services to add nodes or their references in the Address Space.

3 Requirements

Modbus devices usually provide control systems with process data from a remote terminal unit (RTU) on the field level (level 0). Another use for Modbus might be between PLCs of two different manufacturers that needs to exchange information. Usually some of the process data in the PLCs or Modbus devices are valuable in process development or quality control and if it is, that data needs to be stored somehow. To store this data, it is usually transferred to the higher levels of the automation pyramid (figure 1) to MES, MOM or ERP, depending on the type of data to be stored.

As stated in chapter 1, the plant floor level automation and the enterprise level IT infrastructure should be separated from each other as a good practice and to avoid problems with bandwidth. To keep the security level as high as possible, it is a good idea to use a firewall between the networks and only explicitly allow traffic through them. There are several factors on why separation of industrial and enterprise IT networks is preferable. The enterprise network might perform network scans that reserve bandwidth in critical situations [28]. That is one of the main reasons to separate the industrial or automation network from the rest of the enterprise network. Another reason to separate the networks is the fact that a production environment with machines like electrical drives or welding machines create a lot of noise compared to an office environment. This creates higher noise handling requirements on the industrial networking hardware with industrial standards often requiring 10 times the noise handling capabilities of regular enterprise networks [29].

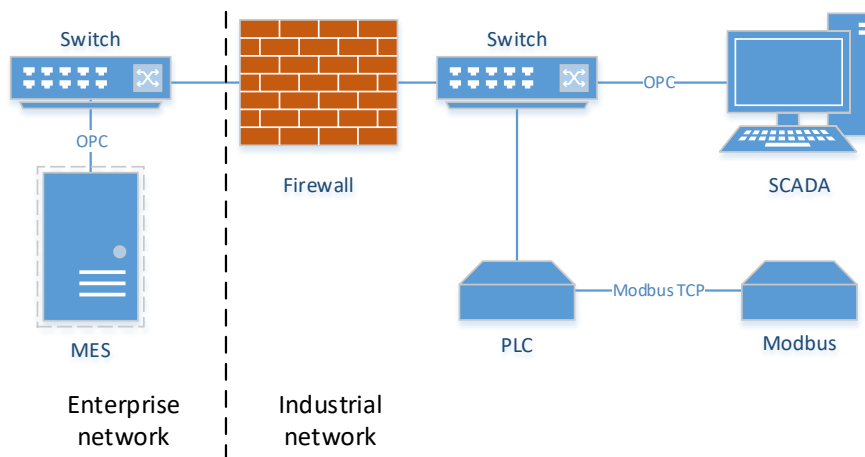


Figure 18: Modbus data to MES.

Modbus TCP devices are built on standard TCP/IP and can thus provide data over the public internet (WAN). But as the Modbus protocol is open and insecure, it needs to be encrypted or isolated to not be vulnerable to attacks or misuse. Usually the data from a Modbus device is read by a PLC that provides the data to a SCADA PC. The SCADA PC then provides the data, for example, over OPC DA to the

MES as illustrated in figure 18. As the old OPC uses DCOM, transferring data between two networks requires allowing traffic over a wide range of TCP ports. By default these ports are 135 and 1024 through 65535 [30]. Allowing traffic through the firewall on all these ports will create an obvious extra security risk and should therefore always be avoided if possible.

A very good solution is to provide this data over OPC UA instead as it only requires allowing communication through the firewall over a single TCP port. OPC UA not only communicates over one TCP port but in addition it can sign and encrypt the data as well, making it a very secure and agile way of transferring data.

3.1 Use cases

There are two main use cases in which the OPC UA Modbus Server application could be used. The first use case is reading process data from Modbus devices on the field level and the second is a PLC reading data that the OPC UA Modbus Server provides. The use cases themselves may have several variations but for simplification only one variation per use case is presented.

3.1.1 Reading production data to MES

The most likely use case scenario is that a user wants to transfer process data from Modbus devices to a Manufacturing Execution System (MES). The Modbus TCP device is connected directly to an Industrial Ethernet network as in figure 19 or sometimes through a Modbus RTU to TCP Gateway. The MES is located in a separate network and the two networks are connected through an industrial firewall. The OPC UA Modbus Server application is installed on a server in the industrial network so that it can access the Modbus devices. In figure 19 the OPC UA Modbus Server is installed on the SCADA server. The OPC UA Client in the MES connects to the OPC UA server through a firewall.

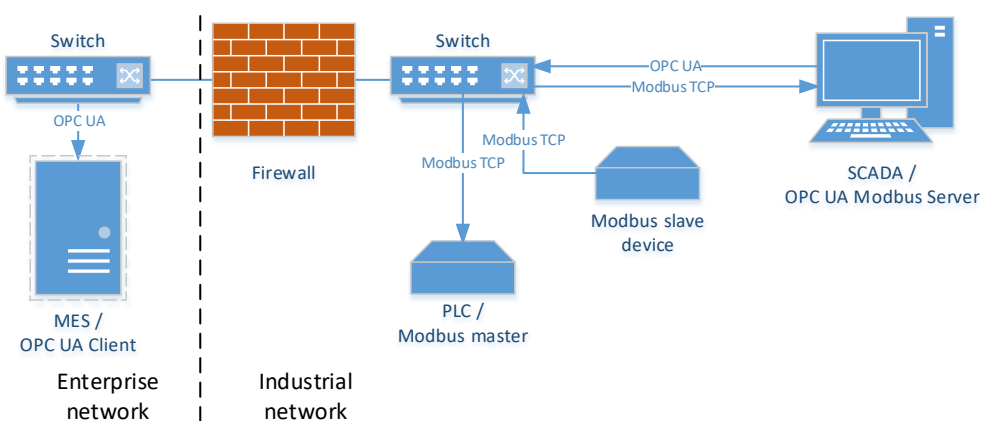


Figure 19: Modbus data to MES.

OPC UA makes it possible for the user to use several features that Modbus does not provide. For example, making the OPC UA client subscribe to the values in the OPC UA Modbus Server the traffic through the firewall can be minimized. By using a subscription the data is only transferred on value changes which usually is a lot more infrequent than cyclical polling. In cyclical polling the data values will be transferred on a regular interval regardless of if it has changed or not. This means that if a value does not change for a day, the same data will still be transferred over and over again. The OPC UA Modbus Server can also be used as a gateway so that it reads data from several Modbus devices and provides the data from all of them over OPC UA on a single connection, meaning it removes the need for creating several connections through the firewall.

3.1.2 Providing PLC with process runtime parameters

The other use case is that a user wants to provide configuration parameters to a PLC from MES. The PLC illustrated in figure 20 already acts as a Modbus master as it continuously reads process data from Modbus slave devices. Thus the OPC UA Modbus Server application should provide data as a Modbus slave device from which the PLC then can read the necessary runtime parameters. The OPC UA Client in the MES connects and writes data to the OPC UA Modbus Server. The OPC UA Modbus Server then provides the data over Modbus TCP, or any other of the implemented Modbus protocols, to Modbus master devices like the PLC.

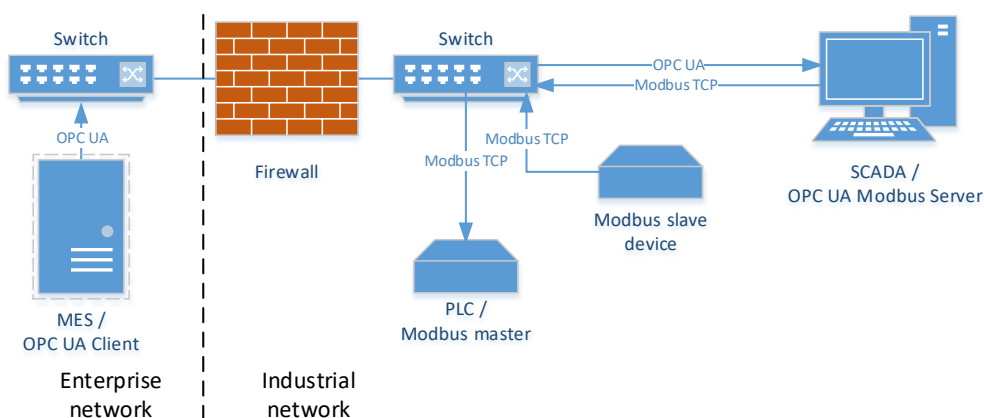


Figure 20: Modbus data from MES to PLC.

3.2 Evaluating OPC UA enabled Modbus applications

Some existing OPC UA Modbus Servers are tested to establish a good starting point for the application design. They are also important for later on being able to compare and evaluate the outcome of the developed application. OPC UA enabled Modbus applications or devices are very rare on the market but do exist. To get a good

baseline and several different viewpoints, four OPC UA enabled Modbus applications are tested.

As the evaluation quickly would get too comprehensive only two use case tests, that are considered the most important ones, are performed on the applications. The tasks being: configuring the OPC UA server and adding a Modbus device with one holding register item.

3.2.1 KEPServerEX

The KEPServerEX is a connectivity platform that provides a single source of automation data [31]. The KEPServerEX is a module based application with a wide range of different modules where users can choose to purchase only the module(s) necessary for them. It supports some of the most current interfaces like MQTT, Splunk and OPC UA as well as older ones like Modbus, Siemens Industrial Ethernet or OPC DA.

Adding a Modbus device

Adding a device to KEPServerEX is done by first defining a communication channel. The channel setup wizard seen in figure 21 guides the user through configuring the communication channel. It specifies which communication interface (NIC, COM, etc...) should be used and its parameters. By selecting *Modbus TCP/IP Ethernet* as the channel type, all parameters are assigned default values that work in most cases.

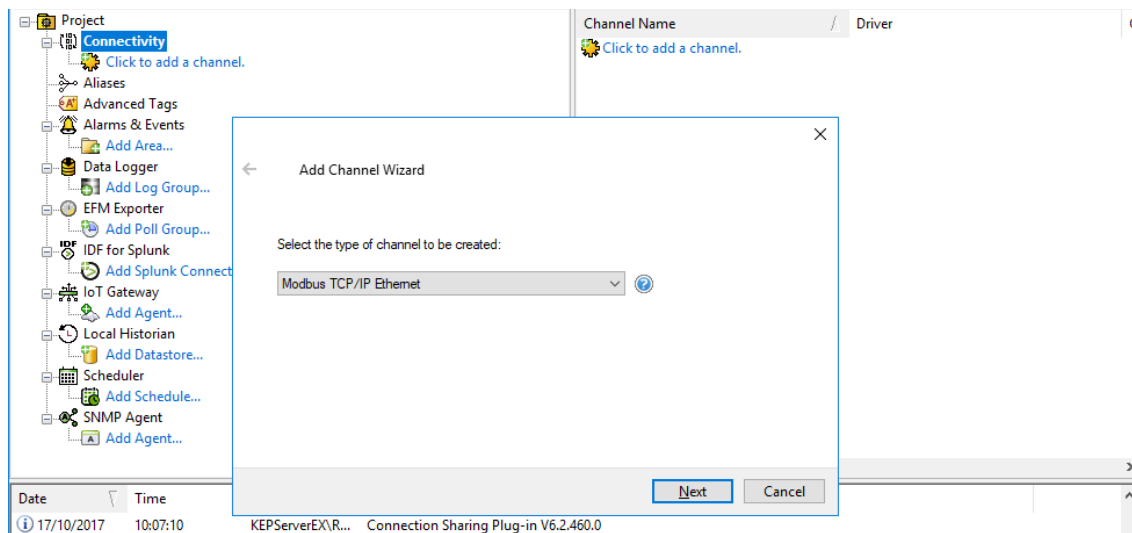


Figure 21: Channel wizard of the KEPServerEX configuration.

When the communication channel is defined, devices can be added to it. In figure 22 the add device wizard is presented. In this wizard the default values of all the parameters can be used. The only necessary modification is to set the IP address of the Modbus device.

The adding of a data item is done by right-clicking on the newly created Modbus device and selecting *New tag*. Then the tag is named and optionally given a description

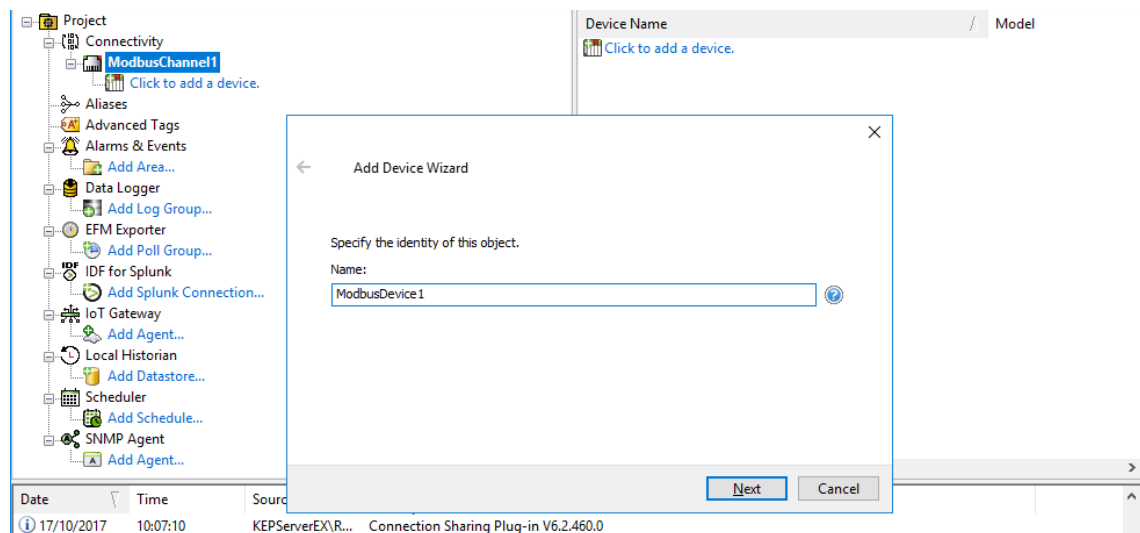


Figure 22: Device wizard of the KEPServerEX configuration.

as well. The important part in the tag configuration is the data properties, more specifically the address and the data type. As figure 23 shows, the KEPServerEX tag data properties use the old way of addressing, e.g. the first holding register being 400001. Which data type to use completely depends on what value type the Modbus device provides. In this case the type is a 32-bit unsigned integer which is referred to as *DWord* in KEPServerEX.

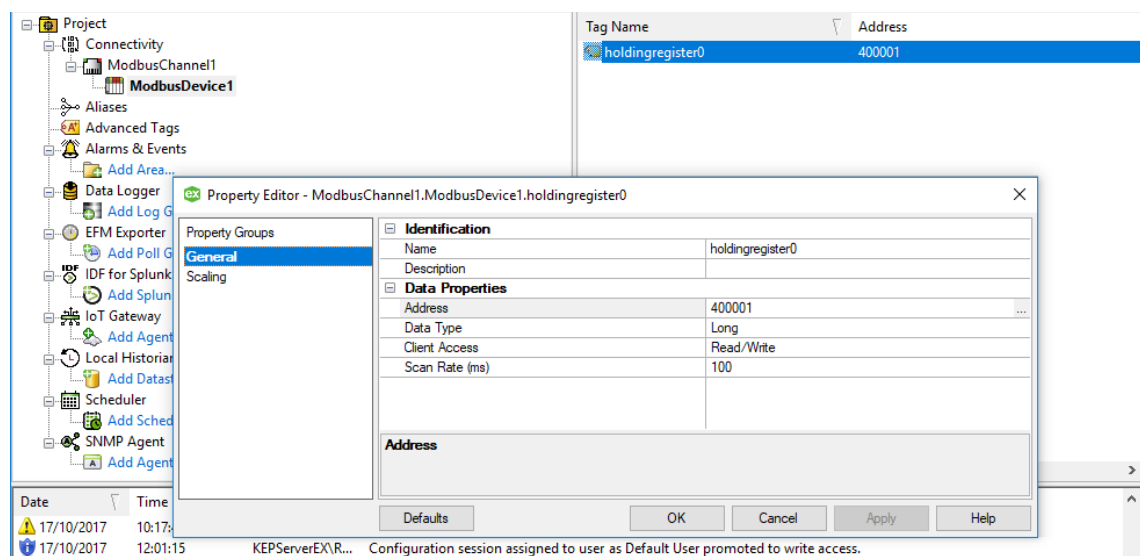


Figure 23: Adding a static tag to the KEPServerEX device configuration.

Adding a Modbus device to KEPServerEX requires 27 clicks and typing the IP address. Adding a tag only requires five clicks and writing the name and address of the tag, while copying a tag only requires four clicks.

Configuring the OPC UA Server

The configuration view of the OPC UA Server in KEPServerEX is started by selecting *OPC UA Configuration* from the KEPServerEX Administration systray menu. The OPC UA server is enabled and configured by default and should work out of the box. Figure 24 shows the default OPC UA server endpoints where it can be seen that endpoints are only created to localhost.

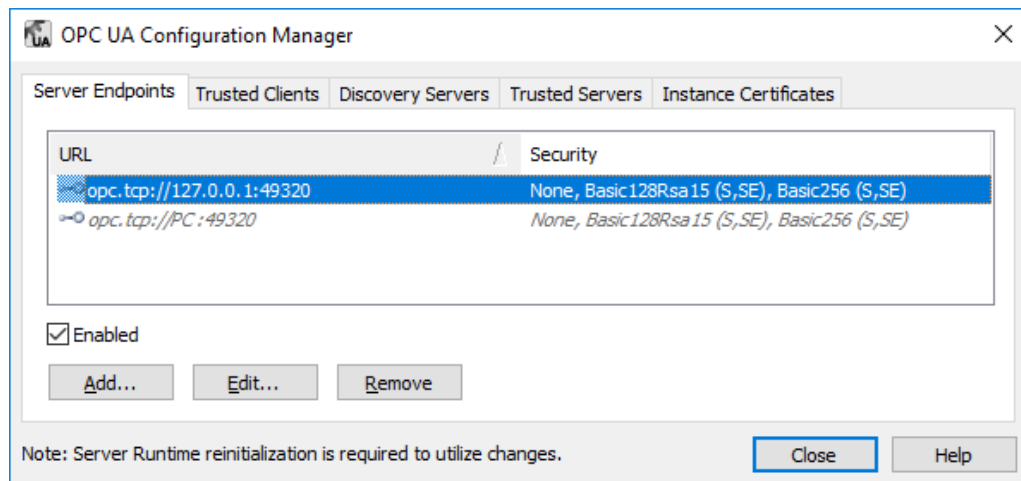


Figure 24: KEPServerEX OPC UA endpoints.

To be able to reach the OPC UA server from the network, the NIC and IP of the machine needs to be added as a server endpoint. When adding an endpoint definition (figure 25), selection of a network adapter will automatically assign the IP address, or more correctly, the endpoint URL.

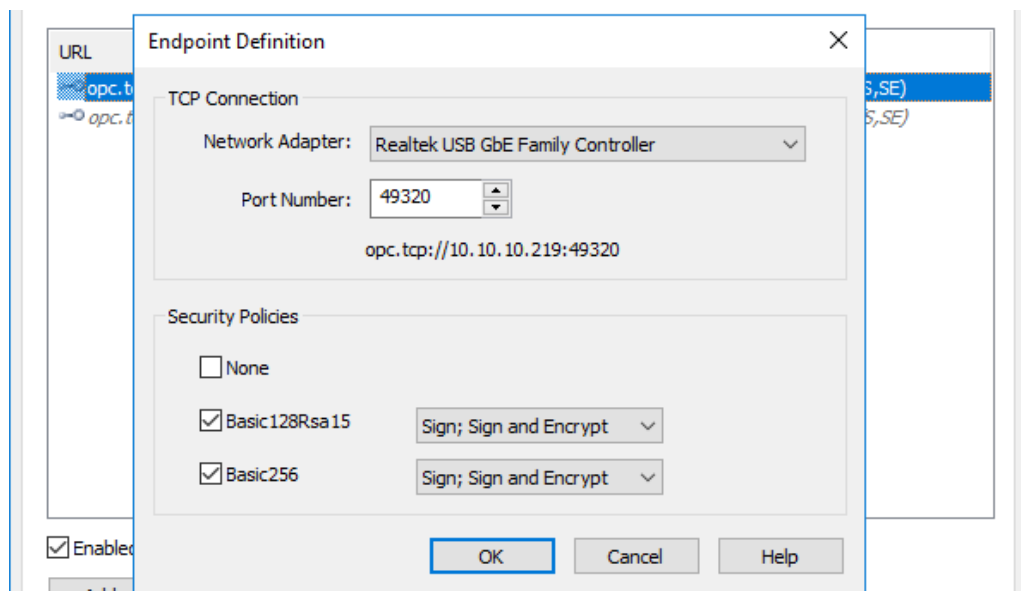


Figure 25: KEPServerEX OPC UA endpoint configuration.

After creating the configuration it can be tested by connecting an OPC UA Client to the KEPServerEX OPC UA Server. The Modbus value can be seen in figure 26 where the node holdingregister0 has a value of four and the data type *UInt32*.

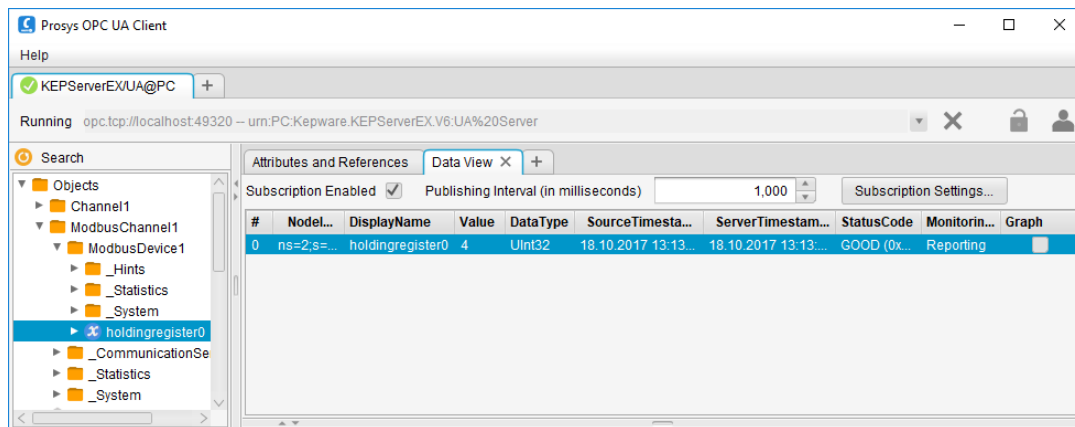


Figure 26: Viewing KEPServerEX value through an OPC UA Client.

Overall impressions

The KEPServerEX has a slightly steeper learning curve in the beginning than the other evaluated applications. The server includes a lot of features that might overcomplicate things if reading Modbus values is the only desired function. As the application is modular, the user can choose to only purchase the necessary modules, but it does not make the configurations any less complicated.

The wizard for creating new channels and devices require a lot of work but is fairly easy to use as the default values can be used in most cases. The separated connector configuration mainly feels unnecessary and overcomplicated. Although it might give the user more configuration possibilities, the average user will have no practical use for it. The tag configuration for a Modbus register is very straight forward and easy. One downside it has is that it uses the old way of naming register addresses. Another downside is that byte, word and dword swapping is only possible on device level, meaning no single specific tag can have a byte-swapped value.

The OPC UA Server configuration is very easy to set up as most of it is done automatically. Worth mentioning here though is that it is very confusing that the OPC UA server settings can be found in two separate places: the project properties in the *KEPServerEX 6 Configuration* application and in the *OPC UA Configuration* application. The OPC UA server is run as a Windows service separately from the configuration application, which means it can easily be left running in the background after the configuration is done. Overall it can be said that if the user can find the time to get familiar with the configuration environment this application could be a suitable solution for connecting to devices from some of the big automation device providers like Siemens, ABB, Allen-Bradley or Omron as they are widely supported.

3.2.2 Cogent DataHub Modbus OPC Server

The Cogent DataHub can collect data from several different sources and protocols into a single data set [32]. It provides communications over OPC DA, OPC UA, Modbus, ODBC and many more. The DataHub is a feature or module based application where the user can choose to license the necessary features only. The DataHub Modbus OPC Server is a feature pack that enables real-time communication between Modbus TCP slaves and any OPC enabled application [33].

Adding a Modbus device

To add a Modbus device to the Cogent DataHub, the Modbus option of the properties window is selected and then the *Add Slave* button is clicked. This opens the Modbus slave configuration window seen in figure 27. Here the only mandatory setting to create a connection is to assign the IP address of the device to connect to, the rest of the settings can be left as default.

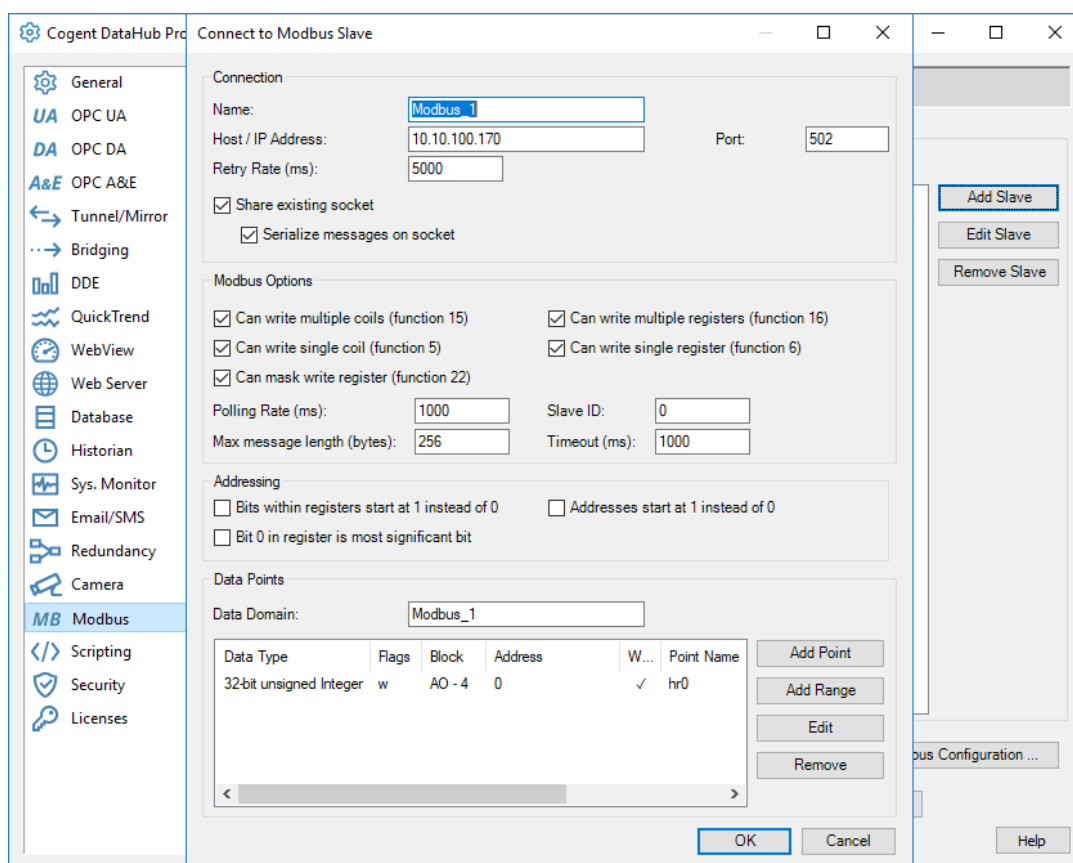


Figure 27: Cogent add Modbus slave device view.

The DataHub Modbus devices call the Modbus data items *data points*. A data point is added by clicking on the *Add Point* button, also seen in figure 27. In the data point configuration window the Modbus data type is selected. The data value type is configured in a slightly different way than in the other evaluated applications,

as can be seen in figure 28. Instead of directly stating the data type like Byte, Word or UInt, it is done with three different settings: Number Type, Encoding and Sign, which tells if the data point is an integer, float or string, if it is a 16-, 32- or 64-bit value and if it is signed or unsigned.

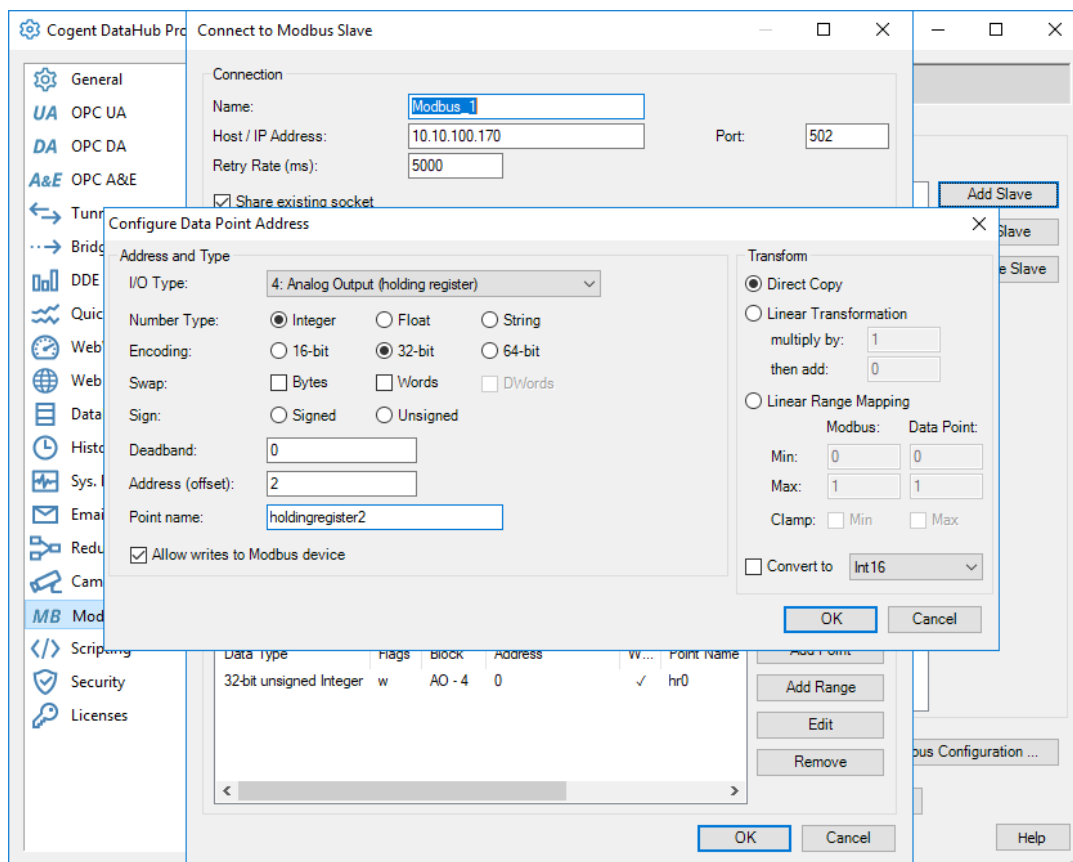


Figure 28: Cogent Configure data point view.

Adding a Modbus device only requires four clicks and typing the IP address. Adding a tag requires nine clicks and setting the name and address. Copying a tag only requires three clicks.

Configuring the OPC UA Server

Configuring of the DataHub OPC UA server is done in the OPC UA option tab as in figure 29. Here the user can define and configure both OPC UA clients to connect to and the server that will provide the data of the application over OPC UA. It offers the most usual server configuration parameters on the same tab. Here the user can easily choose the protocol, security and user token policies or they can manage certificates, like accepting or rejecting a client certificate. The server status also provides a list view of active sessions where the user can view who, from where and usually even what application is connected to the OPC UA server.

After the OPC UA server is configured, it is reloaded by clicking the *Reload Configuration* button and is then available for clients. Then to verify that the OPC

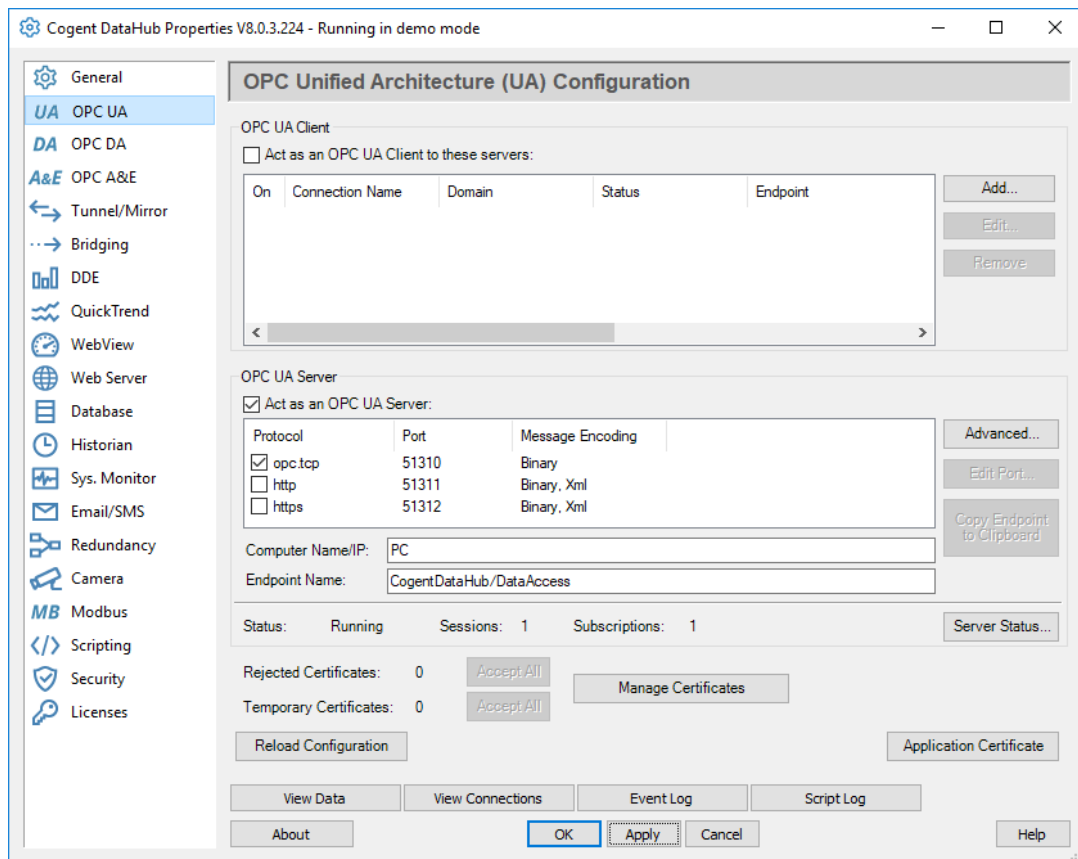


Figure 29: Cogent Configure OPC UA Server.

UA server is available an OPC UA client is used to connect to the server. By browsing to the Modbus device register and adding it to the data view, the value can then be seen. As figure 30 shows, the first holding register has a value of four and a data type of *UInt32*.

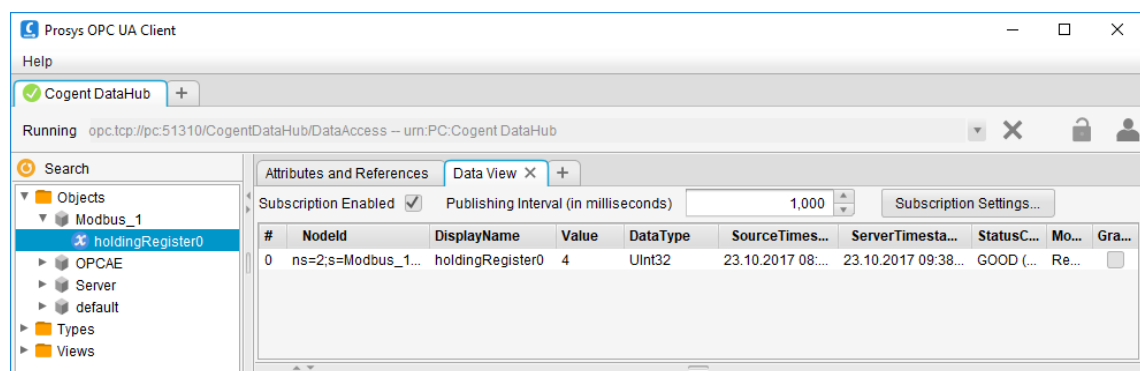


Figure 30: Viewing Cogent DataHub value through an OPC UA Client.

Overall impressions

It is very easy to start using the Cogent DataHub. The settings are organized in a very clear way and are easy to find even without consulting the user manual. The server can be set up and running very quickly even if a user never has used the application before. It has a lot of features available on top of Modbus and OPC UA that some users might count as a big plus.

The Modbus driver configuration makes adding devices very quick and easy. Adding a data point is quite quick as well, but as stated earlier, setting the data type of a data point could be done with one parameter instead of three. A big plus for this application is that it does not use the old way of addressing the Modbus data types and it enables the use of data point specific byte swapping and adding multiple data points at once.

The Cogent DataHub is not installed as a Windows service by default, but can be installed quite easily through an application called *Service Manager* that is provided in the installation package. In general it can be said that if rapid deployment is necessary this could be a very suitable choice. The downside to this is that there is a lack of connections to other protocols like Siemens Industrial Ethernet.

3.2.3 Softing dataFEED OPC Suite

The Softing dataFEED OPC Suite is a pure all-in-one solution for use on Windows based computers [34]. It provides access to controllers from several of the big automation providers like Rockwell, Siemens and B&R [35]. In addition to these it also provides connection over MQTT and data bridging.

Adding a Modbus device

Devices to connect to are located under the *Data Source* tab of the configuration window as seen in figure 31. To add a Modbus slave device, select the Modbus data source and click on add new data source. This opens a wizard that guides the user through adding a device.

The wizard provides default values for most of the parameters and the only required parameter is setting the correct IP address of the device. At the end of the wizard the user is given the option to configure the address space, but this step can be skipped if a connection is all that is wanted by that time. To add a tag to the address space, a user can click on the *Address space definition* icon which re-opens the last step of the device configuration wizard, seen in figure 32. Here a selected tag is being added on the PLC item properties view.

The user can choose to add either a tag or a node to the address space. Nodes are used to organize the address space. For example, holding register tags can be created under one node and input registers under another node. Tags represent Modbus data items. A tag needs to be given a name and an addressing syntax that defines the type of data and the Modbus address. The dataFEED OPC Suite uses an addressing system that is a bit different. For example, the first holding register will have the address HR0. An example of another can be seen in the syntax field in figure 32

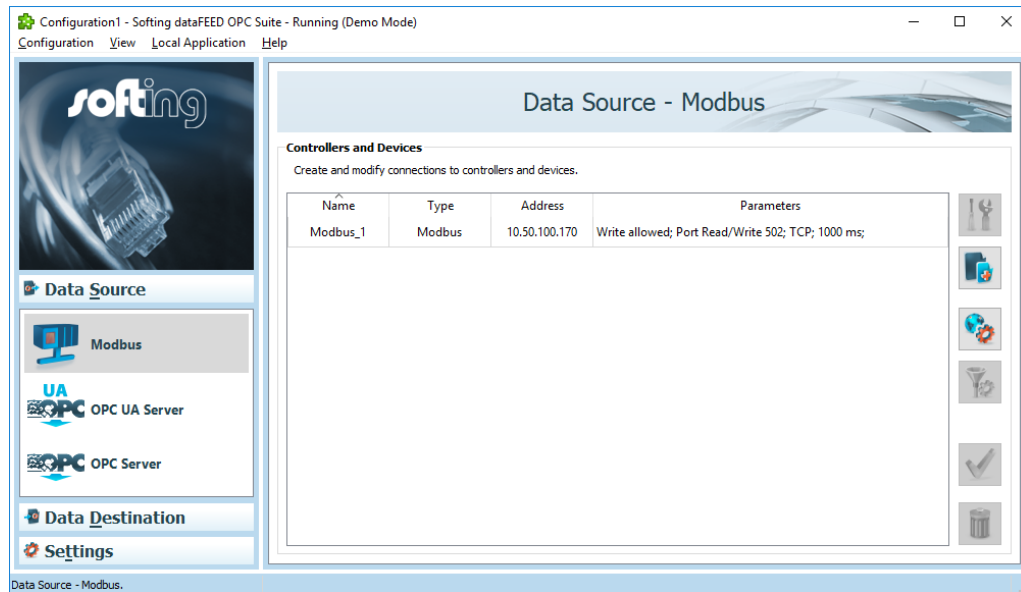


Figure 31: Softing dataFEED configuration of Modbus Data Sources.

as well. To make sense of the addressing syntax the user needs to consult the user manual, where all the supported addressing syntaxes are listed.

Adding a Modbus device to KEPServerEX requires six clicks and typing the IP address. Adding a tag requires seven clicks and typing the name and address. The dataFEED OPC Suite doesn't allow copying of tags.

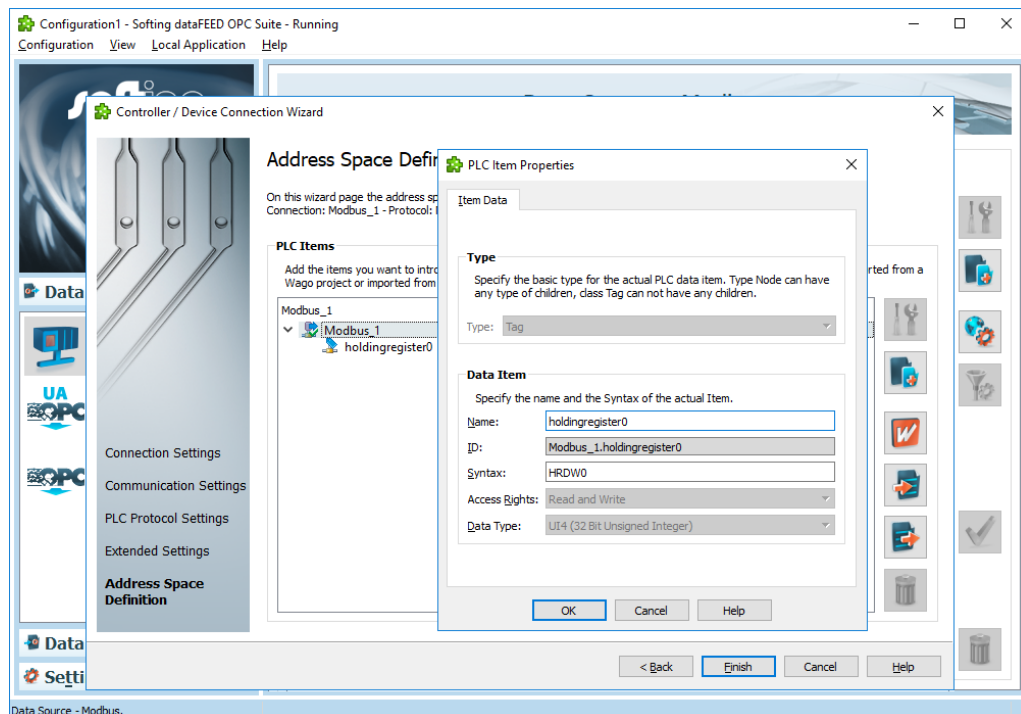


Figure 32: Softing add Modbus tag view.

Configuring the OPC UA Server

The servers or services of the dataFEED OPC Suite that clients can connect to are available under the *Data Destination* tab. The dataFEED OPC UA Server settings are located on the OPC UA Client view seen in figure 33.

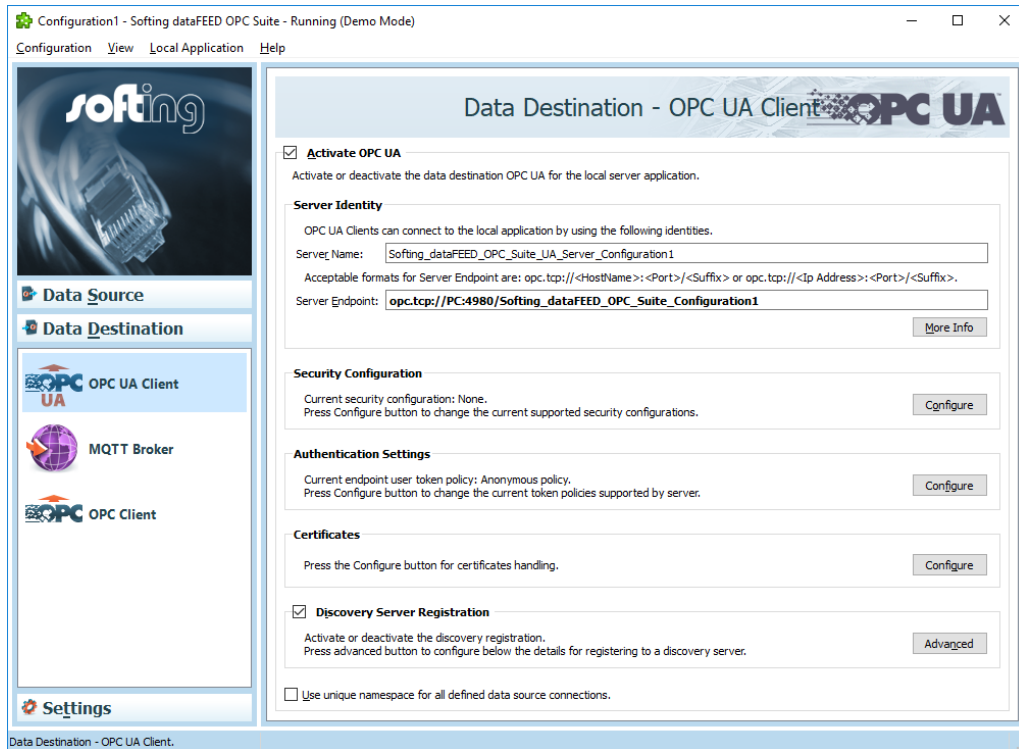


Figure 33: Configure OPC UA Server of Softing dataFEED OPC Suite.

Setting up the OPC UA Server configuration is easy as it will not require any customization at all to work. It is a good idea though to set the security configuration to not allow anonymous or unencrypted connections as the dataFEED OPC UA server allows external connections to it by default. The certificate manager can be reached from this same view as well. The certificate manager allows the user to accept or reject client certificates.

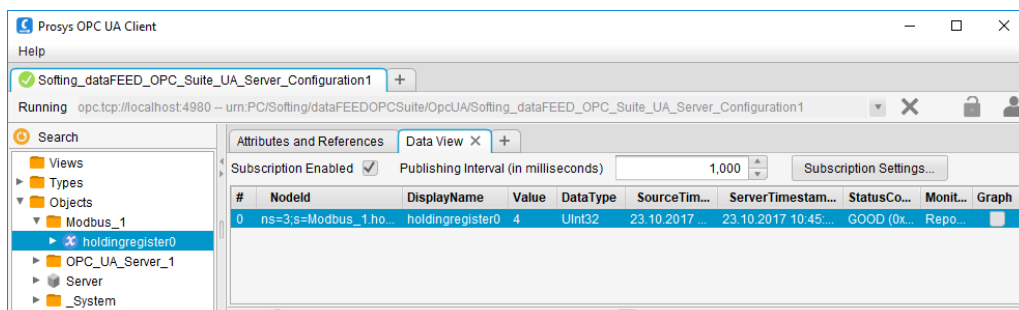


Figure 34: Viewing Softing dataFEED OPC Suite value through an OPC UA Client.

After the server has been configured it is tested by connecting an OPC UA client to it. The Modbus device will have the same structure in the OPC UA Address Space as defined in the previous section where the Modbus device was created. The value in figure 34 shows that the Modbus device is available in the OPC UA Address Space and that it has a value of four and the data type *UInt32*.

Overall impressions

The Softing dataFEED OPC Suite is easy to install and configure. Setting up local clients that connect to external servers are fairly easy as well. Configuring the local OPC UA server is easy, once the configuration view is found. It is quite hard to find the local OPC UA servers configuration view as it is located under Data Destination and OPC UA Client, even though it is a server. The buttons in the Modbus Configuration Address Space Definition wizard uses icons and it is hard to interpret what each of them mean, but fortunately a tooltip explanation is shown while hovering the mouse pointer over the button.

The dataFEED OPC Suite Modbus connector only allows byte swapping on device level and not on individual data items just like the KEPServerEX. The application can easily be configured to be started as either an application or a service from the systray. In general Softing has succeeded in creating a simple and usable application despite its small flaws. As a plus it has a wide variety of different connectors in addition to Modbus as well.

3.2.4 CommServer OPC UA Server for Modbus IP

The CommServer UA is a package of communication applications for managing data transfer over OPC UA [36]. It does not offer as many different protocols as the other evaluated applications. The CommServer supports wrapping of OPC DA servers and thus also providing OPC data over OPC UA.

Adding a Modbus device

The Modbus devices are defined in the CAS CommServer UA Network Configuration application. Devices are added to the configuration in a similar way as in KEPServerEX by first creating a communication *channel*. The channel then defines a *protocol*, which in this case is Modbus TCP. It implements a *segment* that states the IP address of the device to connect to. The segment then further defines the *port* that connects to the *station* or device. The full hierarchy can be seen in figure 35. The station then implements groups and blocks for grouping and making blocks of the different Modbus data types. The figure also shows that the station defines a group called holding register, inside which a block called Block0 is defined. The block defines a single 32-bit register starting at address 0.

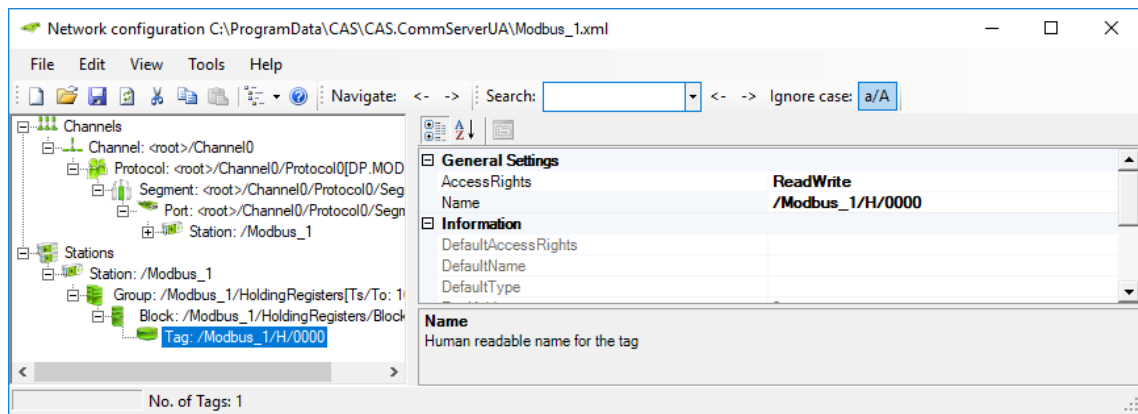


Figure 35: CommServer UA Network Configuration.

Configuring the OPC UA Server

The CommServer UA server address space is created with the CommServer OPC UA Address Space Model designer. To start configuring the server, first a project is created into a solution similarly to for example a C# -project. Then the Modbus address space domain is created by adding some Objects, ObjectTypes or Variables. These nodes are created in the OPC UA Address Space and then the Modbus configuration is mapped to the nodes with data bindings, seen in figure 36. To make the CommServer UA use the created address space the configuration file CAS.CommServerUA.exe.config needs to be modified to point to the uasconfig file built with the Address Space Model Designer.

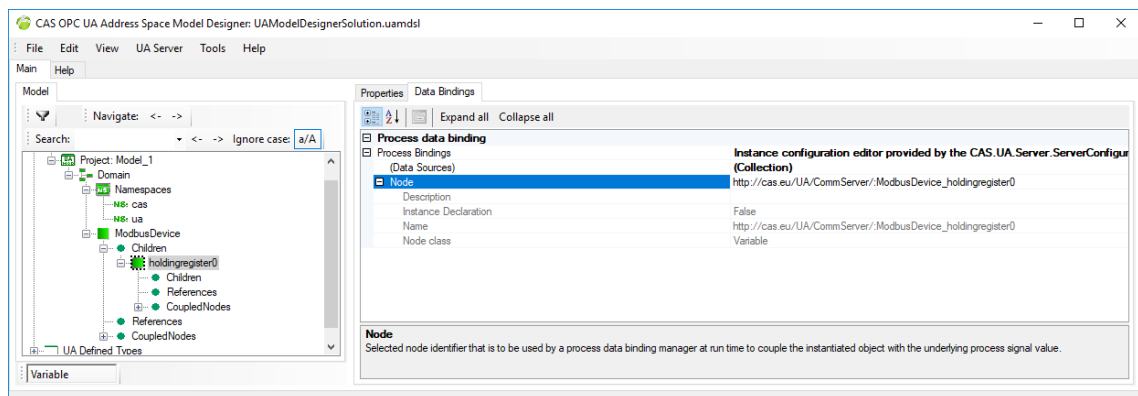


Figure 36: CommServer OPC UA Address Space Model Designer.

After configuring the UA server, it is tested by connecting an OPC UA client and browsing to the node to read. As figure 37 shows, the configured Modbus device can be found in the address space and the configured holding register can be found under the device. It also shows the value is of the data type *UInt32* but there is no value for the holding register. The status code of the value is *BAD* and that means the value can not be read at all.

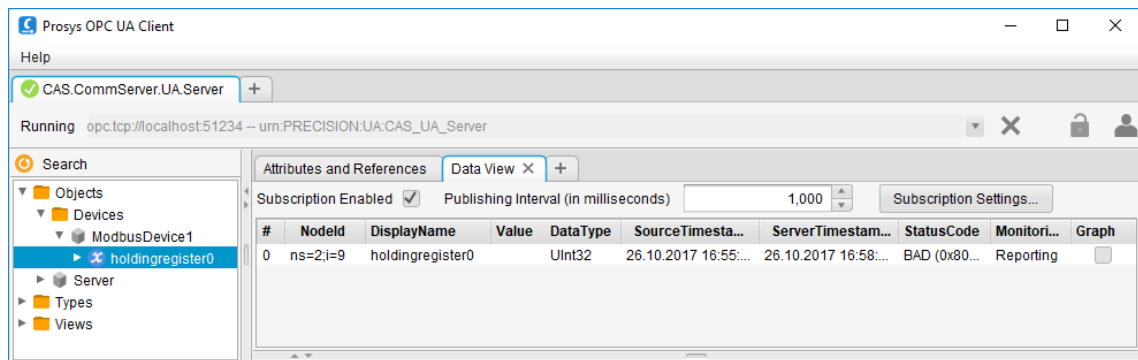


Figure 37: Viewing CommServer UA Server value through an OPC UA Client.

Overall impressions

The application is very hard to use as it consists of a Network Configurator, OPC UA Address Space Model designer, CommServer UA Server and lots of configuration files. Even though both the Modbus configuration and the OPC UA configurations were successfully created, the value could not be read. Debugging this issue is very hard and time consuming as there are so many configuration files that have references to each other and the user manual does not provide any useful information on that matter.

This application uses a more OPC UA-like approach by making the user create an Address space model of the Modbus device domain. It is good that some of the development is heading in the direction of enabling address space modelling. But in this case it is not a good approach as it seriously complicates the application configuration for users not that experienced with OPC UA. Using this application package efficiently would require the user to be trained by a professional.

3.2.5 Summary

The evaluated applications are quite easy to work with although some may require a bit more time to get familiar with than others as some of the settings might not be where they are expected to be. The only exception being the CommServer which requires more in-depth knowledge due to its complexity. To find requirements for the OPC UA Modbus Server to be developed, missing functionalities or usability issues are concentrated on. The adding of Modbus devices and tags might have room for improvement as it feels like there are some unnecessary steps that could be skipped. The configuration of the OPC UA servers are mostly automated which doesn't leave much room for improvement.

Some flaws that all the evaluated applications have in common are that none of them can be run in Linux or Unix based environments. All of the applications allow the creation of Modbus masters that connect to the Modbus slave devices but none of them allow creation of Modbus slave devices for Modbus master devices to connect to. Last but not least none of the applications create Modbus devices in a OPC UA like manner by defining a domain specific information model.

4 Design

4.1 Design decisions

The main goal for the OPC UA Modbus Server is to provide a solution for the two use cases in section 3.1. The OPC UA Modbus server should be designed by concentrating on usability from an inexperienced users point of view and on OPC UA fashioned design on the technical side. The inexperienced user is defined as an IT or automation professional without deeper knowledge of either OPC UA or Modbus. This user should be able to get the application up and running without any greater adversities.

The OPC UA Modbus Server requirements are defined with the evaluated applications in section 3.2 as a baseline. They revealed several requirements of both existing and missing functionalities that should be implemented. The goal is to match or outperform them in the cases of usability and compatibility. With usability focusing on ease of use and compatibility on supporting as many Modbus device functionalities as possible.

The OPC UA Modbus Server will be based on the existing Prosys OPC UA Simulation Server. The Simulation server already implements an OPC UA server and thus it is only necessary to implement additional functions that will map Modbus to it. As the Simulation Server is built based on Prosys OPC UA Java SDK, it will naturally need to be used in this application as well.

The OPC UA Modbus server will provide the most common data types used with Modbus. These data types can be seen in table 10. The amount of supported data types is kept to only a few but can be expanded to cover more data types if necessary.

Table 10: Data types of the OPC UA Modbus Server.

Modbus data type	OPC UA data type
INT	Int16
UINT	UInt16
DINT	Int32
UDINT	UInt32
REAL	Float

Byte and word swapping is an important feature to be included in the application as Modbus device manufacturers may use different approaches, especially in the word ordering. Byte and word swaps should be available at least on device level as in most cases the swaps are similar throughout all the registers. The option to enable swapping on item level is not a major increase in workload and will give users the benefit of freely modifying incompatible values and thus it will be implemented in the application.

As section 3.2.5 states, none of the evaluated applications are available on non-Windows platforms. As Linux is slowly finding its way to the automated control

industry due to its security, stability and reliability and as OPC UA is cross-platform compatible, users should no longer be tied to Windows and its DCOM [37]. Linux distributions enable installation and deployment on virtual headless servers like web servers or on embedded devices in addition to the traditional GUI based. It is clear that the OPC UA Modbus Server needs to support Linux based operating systems as well to give the user more flexibility. As the support can not be guaranteed on every single Linux distribution, it is specified to support at least the most common ones. Them being Debian based and RPM based distributions of Linux. Testing will be done on Ubuntu for the Debian based distributions and on CentOS for the RPM based distributions as these are the most popular distributions respectively.

As one of the use cases in section 3.1 describes, the OPC UA Modbus Server should also provide data as a Modbus slave and not only access data as a Modbus master. This feature is not implemented by any of the applications evaluated in section 3.2 and will give the user more options for connecting different types of Modbus applications over OPC UA.

The OPC UA server endpoints should provide the available security modes and security policies, user authentication methods and certificate handling. These features are already available in the OPC UA Simulation Server and can be used as is without any further development.

The OPC UA Modbus Server should implement its own domain specific Information Model that defines type definitions. The Modbus Information Model should then be used when instantiating the user specified Modbus devices.

As the application can be run on headless servers, it should be possible to configure it without a GUI as well. This should be done by constructing an XML-file that defines the Modbus devices and their necessary settings, the GUI should modify and save the same file so that a user can create a configuration with the application run in normal GUI-mode and then transfer that configuration to the headless server where the application will be deployed.

4.2 Modbus Information Model Design

For the design of the Modbus Information Model the OPC UA specific graphical notation in the OPC UA specification is used instead of the standard UML class diagrams. The notation and how it should be used is described in sections 2.2.2, 2.2.3 and in the OPC UA specification [21]. This part will explain how the Modbus Information Model has been designed.

The complete Information Model of the OPC UA Modbus Server is a combination of the Information Models: Base Information Model, Data Access extension Information Model, Devices Information Model and the custom Modbus Information Model. A complete diagram of the Information Model can be examined in the appendices in figure A1. Only relevant data from the Information Models are presented in the diagrams below, thus trying to reduce the complexity of the full Information Model.

Figure 38 shows the Nodes in the Base Information Model that are used in the OPC UA Modbus Server. These Nodes are either extended to domain specific subtypes or used as such and they define the base for what the Modbus Information

Model is built on.

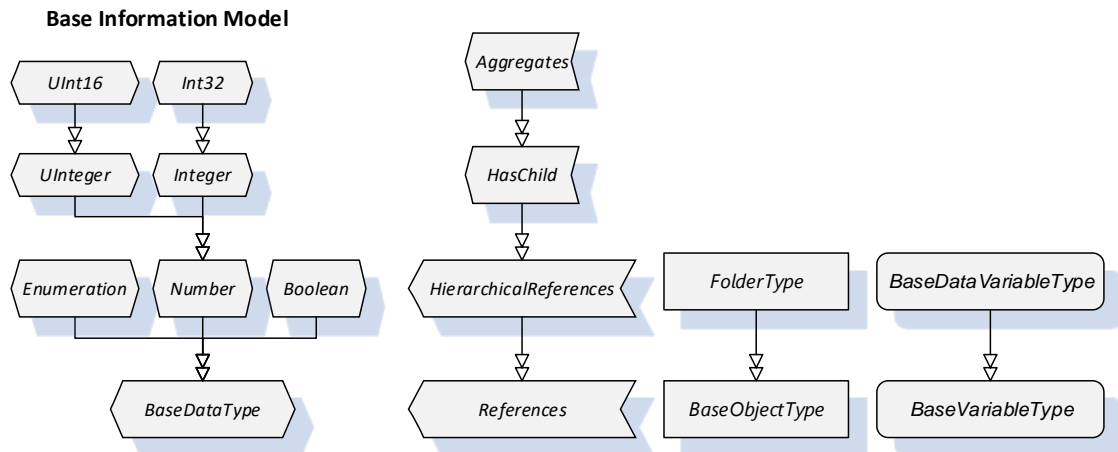


Figure 38: Base Information Model nodes

Modbus devices generally only provide real-time data and not any historical data or events, thus the OPC Data Access extension is used for defining variable types of the Modbus variables. The two VariableTypes *DiscreteItemType* and *AnalogItemType* are both subtypes of *DataItemType*. All of these VariableTypes are defined in the OPC UA Data Access specification seen in figure 39 [38]. A *DataItem* can be described as currently valid, live automation data. Its two subtypes further specify what type of data it contains. The *AnalogItem* provides a value of variable physical quantities like temperature or humidity values while the *DiscreteItem* provides state based values like e.g. opened, closed or moving.

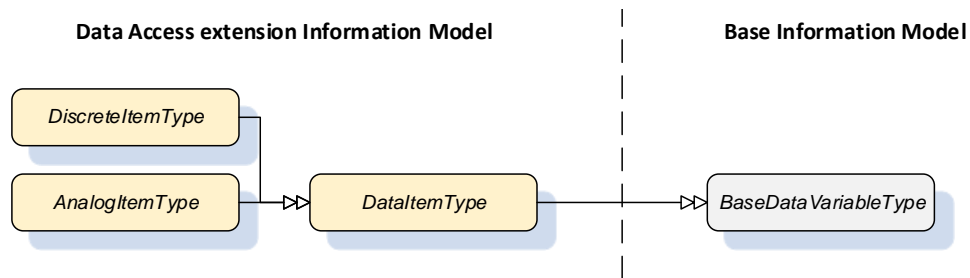


Figure 39: DA Information Model nodes

As the Modbus server is intended for use in industrial environments where PLC:s and RTU:s are used and because the Modbus data usually is provided by devices, it is a good idea to use an Information Model that provides a general type for devices. The OPC Foundation has already created the OPC UA for Devices Information Model for purposes like these. It fits perfectly for this type of application as it is designed to provide a unified view of devices and is chosen as a base for the Modbus devices in the Modbus Information Model [24]. OPC UA for Devices describes

three models which build upon each other, but in this case only one of them is used: the base device model. The base device model includes an ObjectType called DeviceType. Figure 40 shows that this DeviceType is a subtype of the abstract TopologyElementType, which is derived from the BaseObjectType. The topology element is intended to provide its subtypes (in this case the DeviceType) optional objects like for example the ParameterSet Object. In the case that any parameter exists for a TopologyElement, the ParameterSet object is mandatory even if the TopologyElementType definition states that it should be optional [24].

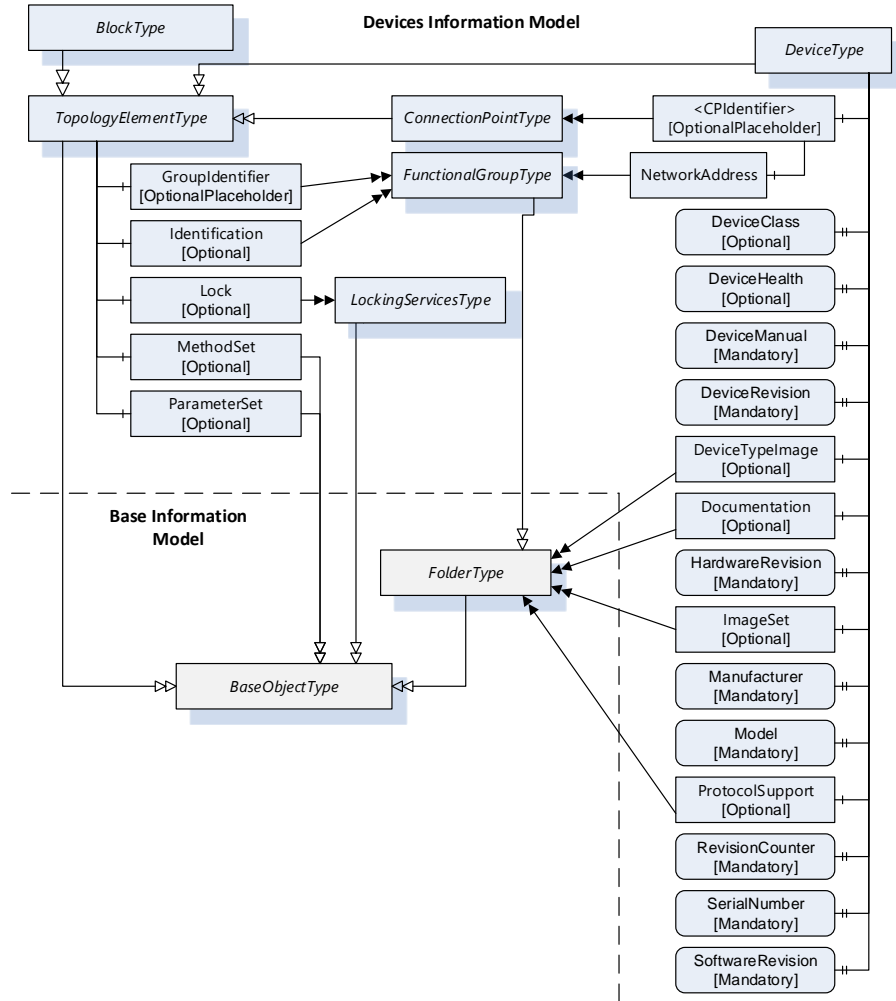


Figure 40: Devices Information Model nodes

The DeviceType is intended to provide information about the device itself, like DeviceHealth, SerialNumber and SoftwareRevision to name a few properties. The DeviceType is as the TopologyElement also abstract, which means that there can't be any instances of DeviceType but only a subtype of it [24].

4.2.1 ModbusDeviceTypes

The DeviceType is extended to a new type called ModbusDeviceType, which is the base of the Modbus Information Model. As stated in section 2.1, there are two types of Modbus devices: slaves and masters. To be able to tell the difference between these Modbus device types, both of them are given their own type definitions: ModbusMasterType and ModbusSlaveType. These are created as subtypes of the ModbusDeviceType. Figure 41 illustrates a very simplified version of the Modbus Information Model to give a general overview of how the types relate to each other. Here the ModbusIoBlockType and ModbusRegisterConfigurationType ObjectTypes that are discussed later on can be seen as well.

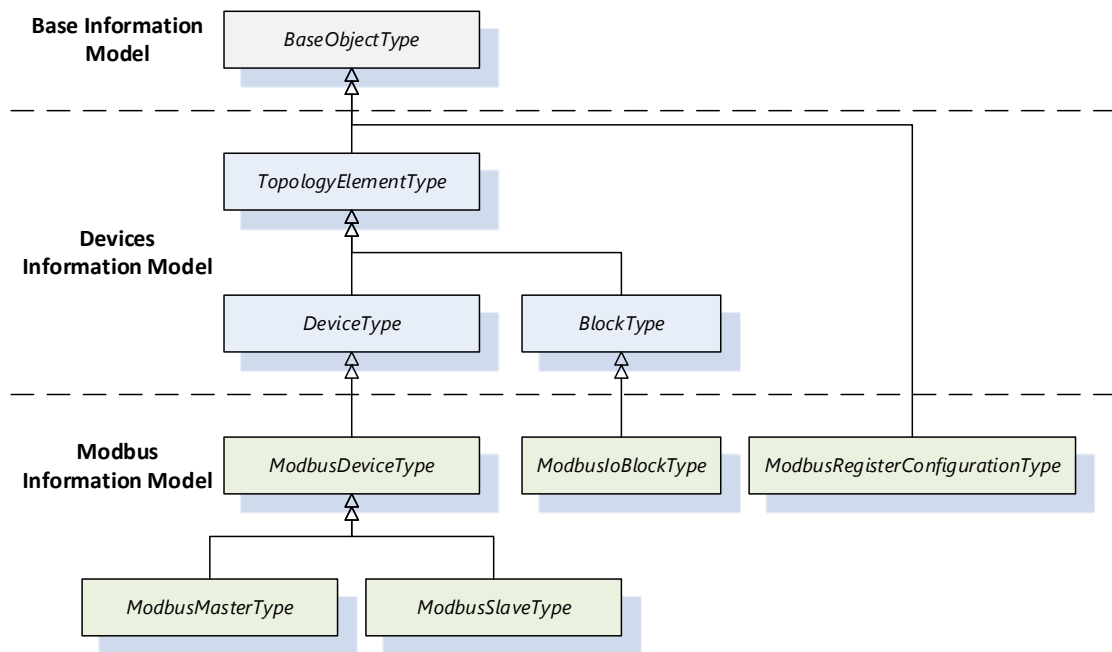


Figure 41: Modbus Information Model nodes

The Modbus Information Model illustrated in figure 42 includes the full architecture and shows how all the parts are linked. It shows the parent nodes and their corresponding Information Models from where the Modbus type nodes are derived. As stated earlier, the ModbusDeviceTypes are subtypes of the abstract DeviceType that is implemented in the Devices Information Model, which is also visible in figure 42. The figure also shows that neither the Modbus master or slave type implement any components or parameters of their own. This is because the ModbusSlaveType and ModbusMasterType is only meant to supply the information on which type of Modbus communication to use. Thus the ModbusDeviceType is the type implementing all the components that are then inherited by the two subtypes.

The ModbusDeviceType implements all available register types as separate blocks. These are all subtypes of the ModbusIoBlockType and are discussed in the next section. In addition to these blocks it also needs to implement the protocol type

and the unit id. The protocol types are defined in the ModbusProtocol enumeration DataType. This Modbus Information Model only defines the four most common Modbus protocol types: Modbus RTU, Modbus ASCII, Modbus TCP/IP and Modbus RTU/IP. The unit id is primarily used to identify Modbus devices on a serial line behind a gateway.

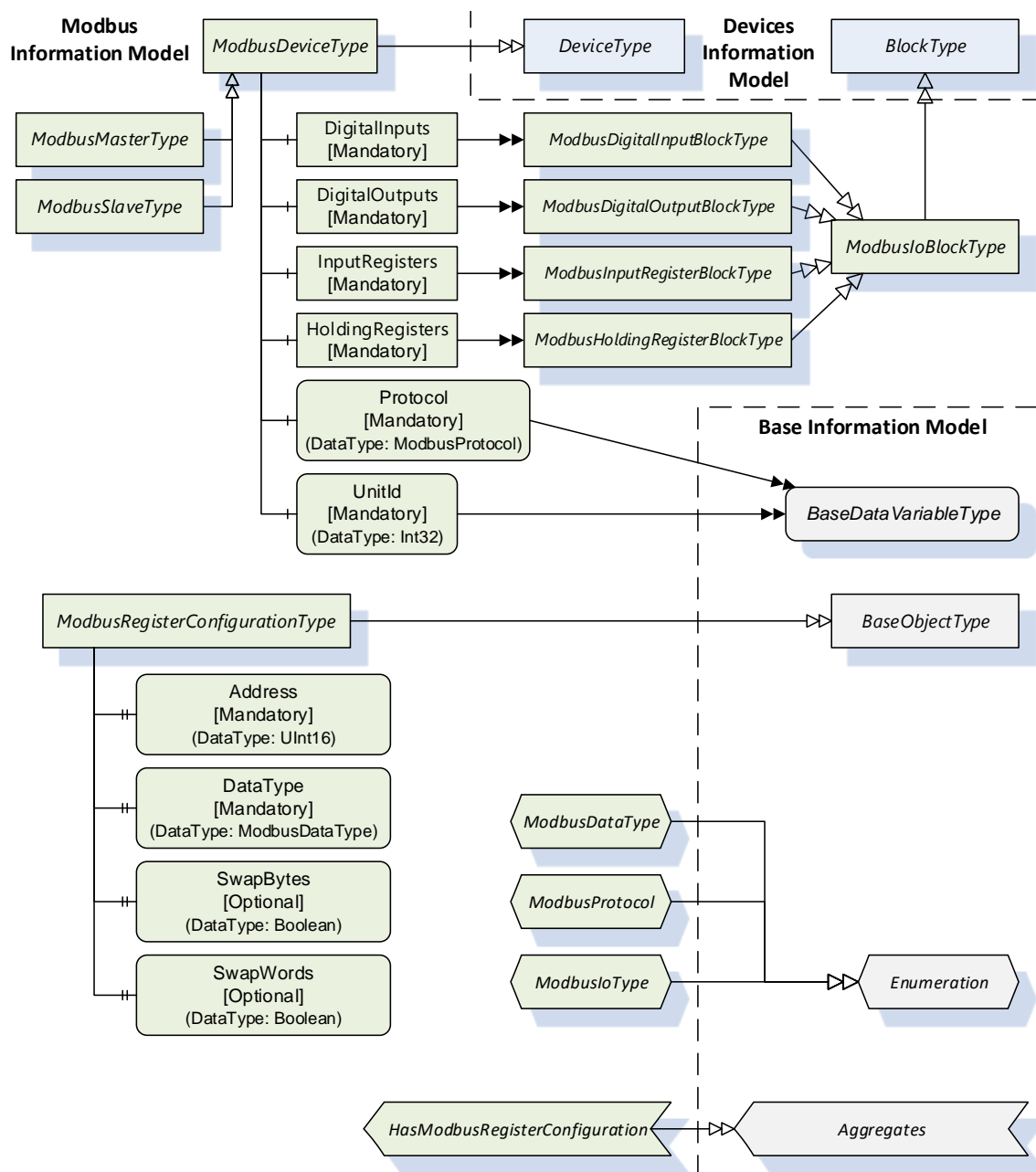


Figure 42: Modbus Information Model.

4.2.2 ModbusIoBlockTypes

The ModbusDeviceTypes always implement each of the available register types in the Modbus Data model listed in table 2, but they might not necessarily define any registers [8]. Each of the register types can be viewed as a separate block of registers and thus four corresponding block types are defined. To group these together, a ModbusIoBlockType is defined and these blocks are created as subtypes of that. These block types can be seen visualized in figure 42.

An example of a ModbusIoBlockType is the HoldingRegisters component of the ModbusDeviceType. It has a HasTypeDefinition reference to the ModbusHoldingRegisterBlockType. This means that it can be concluded that the block contains holding registers by checking the type definition. In a case that the name of the block would for example be CounterValues, it could still be concluded that the registers in this block are holding registers as its type definition would still be ModbusHoldingRegisterBlockType.

4.2.3 ModbusRegisterConfigurationType

The ModbusRegisterConfigurationType is a subtype of the BaseObjectType and is intended to provide the required properties of a Modbus data register. As seen in figure 42 it defines two mandatory properties: Address and DataType, and two optional parameters: SwapBytes and SwapWords. The address parameter is a non-negative 16-bit integer that specifies the start address of the Modbus register. The Modbus specification states that there are 65536 data addresses in each type of Modbus registers and thus 16-bits covers all of them.

The data type defines the ModbusDataType which like the ModbusProtocol, is an extension of Enumeration. The data type options defined in the ModbusDataType depends on the application it is used in. Modbus uses the elementary data types defined in the IEC standard for programmable logical computers [39]. They differ from regular PC data types by using 16-bit values as standard length as opposed to the 32-bit length PCs use. For example an integer is normally 32-bits in the PC world but in the PLC world an integer is only 16-bits. The mapping between ModbusDataTypes and OPC UA data types can be seen in table 11. It also shows which types are implemented in the OPC UA Modbus Server.

The two optional swap parameters enable configuration of byte and word ordering of each data item. As different Modbus device manufacturers might implement their data in either way, these parameters give the flexibility to map the data correctly to OPC UA either way.

4.2.4 HasModbusRegisterConfiguration

The data items, also called Nodes, in a register like the holding registers will be implemented as OPC UA Variables. They will always extend either of the available DataItemTypes. This means they will be assigned a HasTypeDefinition reference to either AnalogItemType or DiscreteItemType. As the OPC UA specification states that a Node can only have one type definition. Because of this a new ReferenceType

Table 11: Mapping of IEC 61131-3 data types to OPC UA data types [40].

IEC 61131-3	OPC UA	Implemented	Description
BOOL	Boolean	yes	1 bit boolean value
BYTE	Byte	no	8 bit "bit string" value
WORD	UInt16	no	16 bit "bit string" value
DWORD	UInt32	no	32 bit "bit string" value
LWORD	UInt64	no	64 bit "bit string" value
INT	Int16	yes	16 bit signed integer value
UINT	UInt16	no	16 bit unsigned integer value
DINT	Int32	yes	32 bit signed integer value
UDINT	UInt32	no	32 bit unsigned integer value
LINT	Int64	no	64 bit signed integer value
ULINT	UInt64	no	64 bit unsigned integer value
REAL	Float	yes	IEEE-754 single precision (32 bit) value
LREAL	Double	no	IEEE-754 double precision (64 bit) value
TIME	Duration	no	Duration
DATE	DateTime	no	Date (only)
TIME_OF_DAY	DateTime	no	Time of day (only)
DATE_AND_TIME	DateTime	no	Date and time of day
STRING	String	no	Character string (variable length)

called HasModbusRegisterConfiguration has to be created so that a Node can be indicated to have a reference to a ModbusRegisterConfiguration. This reference will link a variable Node to its specific configuration object Node.

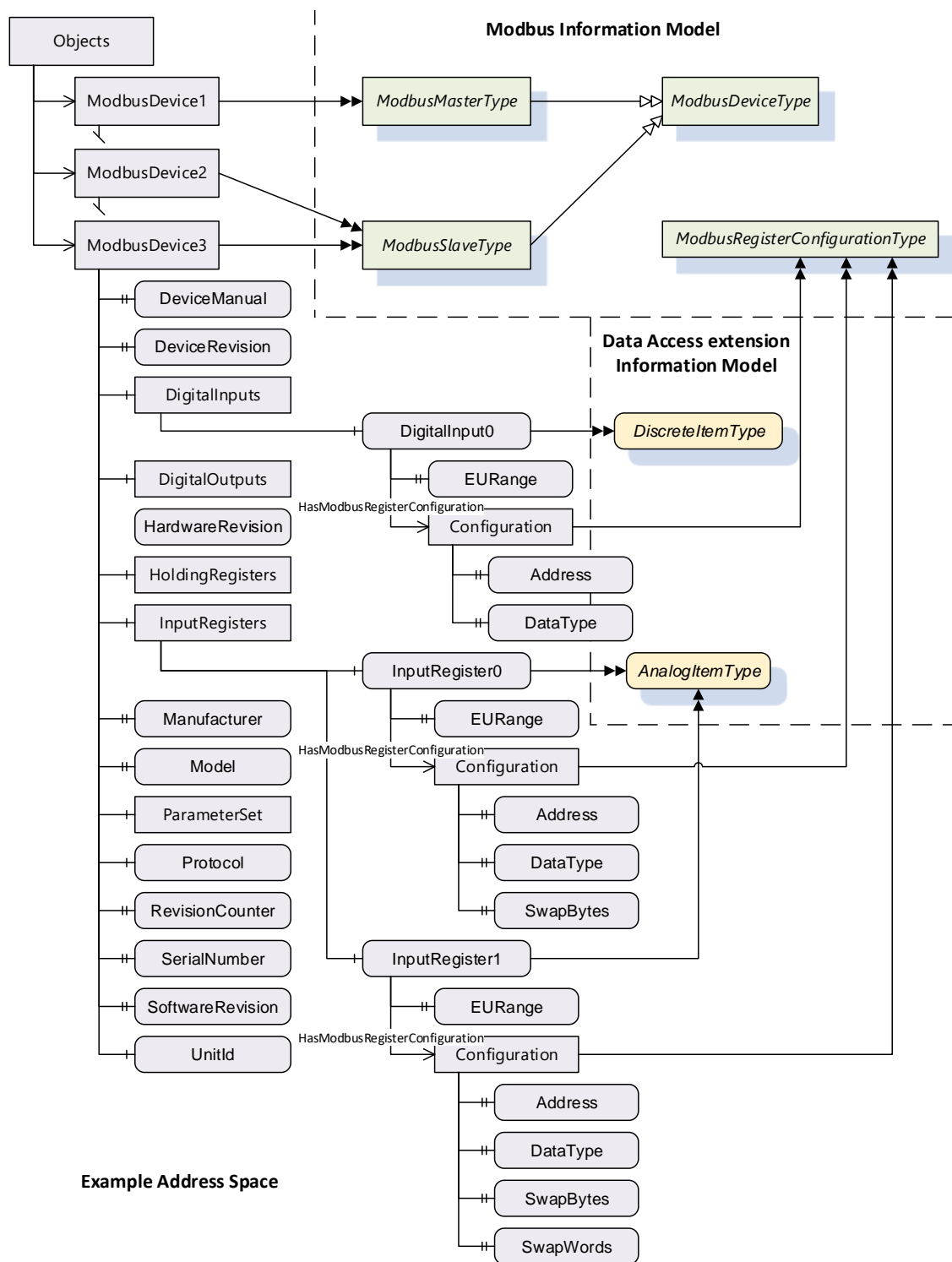


Figure 43: Example of a Modbus Address Space with three Modbus devices.

5 Implementation

5.1 Creating the Information Model

The Modbus Information Model is created with Unified Automation UaModeler, a special tool that "turns design into code". It is used to design the Address Space model of an application and for generating source code for that model [41]. It is built so that it generates well structured and error free code. This software only generates code for C++, ANSI C and .NET based OPC UA Client and Server SDK:s. To generate code for the OPC UA Java SDK that is used in the OPC UA Modbus Server, the Codegen script included in the SDK needs to be used.

Instead of the UaModeler generating source code it can export an UaNodeSet of the Information Model. An OPC UA NodeSet is a generic way to describe a standard Address Space [43]. The UaNodeSet is defined in XML-format following the OPC UA Information Model XML Schema syntax and thus allowing applications to read and process the Information Model [25]. The UaNodeSet defines a set of Nodes, their attributes and references. An OPC UA server that implements an UaNodeSet will have the Nodes, attributes and references in its standard Address Space.

To implement the Information Model, a new project needs to be created with UaModeler. The user generated Information Models always extend the base UaNodeSet and can additionally also extend other UaNodeSets. As stated in section 4.2 the OPC UA for Devices Information Model is used in the design of the Modbus Information Model, so the OPC UA DI NodeSet is included in the project in addition to the base NodeSet.

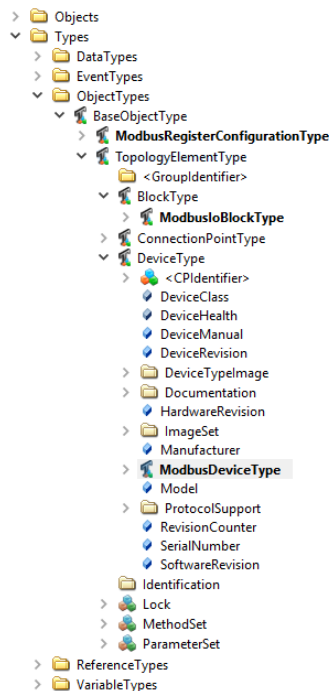


Figure 44: Modbus root Address Space.

The Modbus part of the Address Space is given its own namespace URI to separate the domain specific nodes from the other NodeSets, the URI will be *http://prosysopc.com/UA/Modbus/*. This namespace will define the Modbus domain specific nodes, their attributes and references. After the UaModeler project settings are configured for the Modbus Information Model, modelling can start.

The Information Model is created systematically by first adding the main components and then developing them by adding their details. First off the ObjectType **ModbusDeviceType** is created. It is created as a subtype to the DeviceType as seen in figure 44. In the figure it can also be seen that UaModeler provides a graphical view of the Address Space, consisting of the selected NodeSets in the settings. The new Information Model can be built on top of them by adding Nodes into the Address space. Once they are added to the Address Space, they can be distinguished by a bold title. See, for example, **ModbusDeviceType** in figure 44.

NodeClass	Name	TypeDefinition	ModellingRule	DataType		
> Object	DigitalInputs	ModbusDigitalInputBlockType	Mandatory		+	X
> Object	DigitalOutputs	ModbusDigitalOutputBlockType	Mandatory		+	X
> Object	HoldingRegisters	ModbusHoldingRegisterBlockType	Mandatory		+	X
> Object	InputRegisters	ModbusInputRegisterBlockType	Mandatory		+	X
> Variable	Protocol	BaseDataVariableType	Mandatory	ModbusProtocol	+	X
> Variable	UnitId	BaseDataVariableType	Mandatory	Int32	+	X

< Select NodeClass >

Figure 45: ModbusDeviceType Node configuration.

Following this the **ModbusIoBlockType** is added as a subtype of the BlockType and the **ModbusRegisterConfigurationType** is added as a subtype to the BaseObjectType. Together these three ObjectTypes form the base of the Modbus Information Model as can be seen in figure 41. After these are added their configurations are specified. **ModbusDeviceType** defines the base for Modbus devices, so as figure 42 states, it will implement all the object and variable Nodes as child components. In addition to these, the two Modbus device types **ModbusMasterType** and **ModbusSlaveType** are defined as its subtypes. The implementation is illustrated in figure 45. To be able to add the four IO block objects to the **ModbusDeviceType**, they need to be created. These IO block types will contain Modbus register value Nodes of the respective Modbus data type. The IO block type is only used to provide information about which Modbus register type the Nodes in that block are. This can be seen in figure 46, where the block types do not have any components or properties.

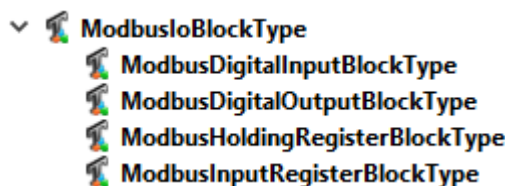
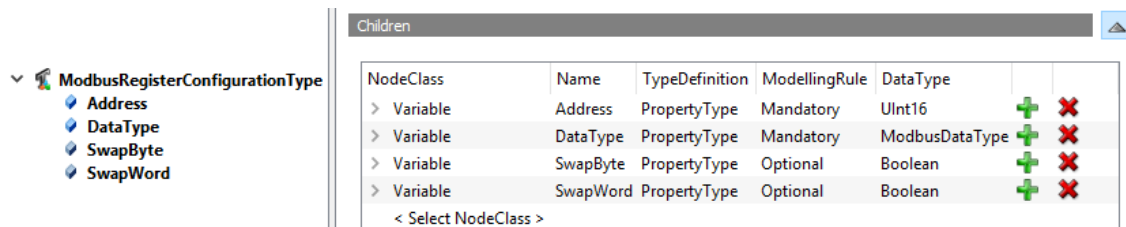


Figure 46: ModbusIoBlockType and its subtypes.

Figure 47 shows the definition of the `ModbusRegisterConfigurationType`. It is used to provide Modbus register configuration properties of a Modbus register value Node. As the figure also shows, the swap variables are optional as, depending on the data type, a swap might not be possible. Now as the Modbus register value Nodes will be defined to have a data access `TypeDefinition` of either `DiscreteItemType` or `AnalogItemType` like figure 43 shows, and the Nodes can't have two `TypeDefinition` references, the new `HasModbusRegisterConfiguration` is created.



NodeClass	Name	TypeDefinition	ModellingRule	DataType		
> Variable	Address	PropertyType	Mandatory	UInt16	+	×
> Variable	DataType	PropertyType	Mandatory	ModbusDataType	+	×
> Variable	SwapByte	PropertyType	Optional	Boolean	+	×
> Variable	SwapWord	PropertyType	Optional	Boolean	+	×

< Select NodeClass >

Figure 47: `ModbusRegisterConfigurationType` definition.

The `HasModbusRegisterConfiguration` `ReferenceType` is a subtype of `Aggregates` and the inheritance path can be seen in figure 48. The figure shows that the `ReferenceType` is similar to e.g. `HasComponent` as they are siblings. As the name hints, the `ReferenceType` tells that a Node that implements this reference has a `ModbusRegisterConfiguration`.

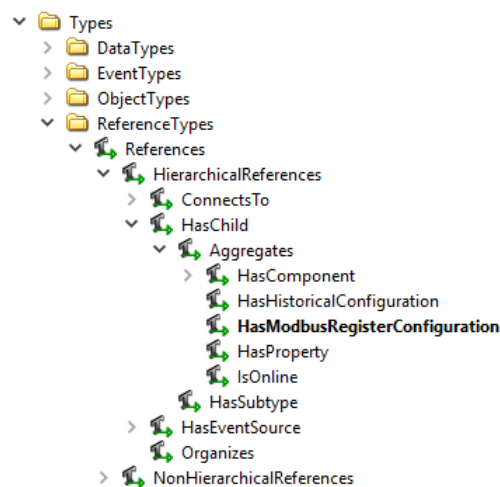


Figure 48: `HasModbusRegisterConfiguration` `ReferenceType` definition.

The `DataTypes` provided by the Modbus Information Model are all subtypes of the enumeration `DataType`. As illustrated in figure 49 there are three Modbus specific `DataTypes`: `ModbusDataType`, `ModbusIoType` and `ModbusProtocol`. The `ModbusDataType` defines an enumeration list of the data types listed in table 10. The `ModbusIoType` defines a list of the register types and is an alternative to the IO block types for providing the register type. The `ModbusProtocol` lists the available Modbus protocol types that can be used. Since the OPC UA Modbus Server only

implements Modbus protocols using Ethernet, only the two options using TCP out of the four that are defined will be available in the application.

Figure 49: Modbus domain specific data types.

When all the parameters are defined and the Modbus Information Model is ready the UaNodeSet XML-file is exported. The NodeSet can be found in appendix B.

5.2 Modbus device configuration

A Modbus configuration is created and stored as an XML file. Modbus master and slave device configurations are separated into their own configuration files. Section 4.2.1 states that Modbus Master and Modbus Slave devices use the same configuration so the structure of the configuration files will be the same for both types. It is designed to be as simple as possible so that a configuration can even be created or modified with a text editor if necessary. An example of a configuration file can be seen in listing 1 below. Upon start of the application, it will parse the configuration files and create the Modbus devices and their respective OPC UA Nodes. Modifications to the configuration will be stored automatically and will take effect the next time the Modbus Server is started.

Listing 1: Example xml configuration for Modbus slave device.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<devices>
  <device name="NewDevice">
    <ip>10.10.10.10</ip>
    <port>502</port>
    <modbusType>TCP</modbusType>
    <unitId>0</unitId>
    <enabled>true</enabled>
    <di>
      <nodes>
        <node offset="0" datatype="BIT" swapBytes="false" swapWords="false">input0</node>
      </nodes>
    </di>
    <do>
      <nodes/>
    </do>
    <ir>
      <nodes/>
    </ir>
    <hr>
      <nodes>
        <node offset="0" datatype="UINT" swapBytes="false" swapWords="false">holding0</node>
        <node offset="1" datatype="DINT" swapBytes="false" swapWords="false">holding1</node>
        <node offset="3" datatype="UDINT" swapBytes="false" swapWords="false">holding3</node>
        <node offset="5" datatype="REAL" swapBytes="false" swapWords="false">holding5</node>
      </nodes>
    </hr>
  </device>
</devices>
```

5.3 Creating the OPC UA Modbus Server

As stated in section 4.1 the OPC UA Modbus Server is based on the OPC UA Simulation Server which is implemented using Java 8. Previous work gives an extensive description of the OPC UA Simulation Server and its architecture, but very shortly put it is based on the Prosys OPC UA Java SDK and some well known Java libraries like Java FX 8 [42].

Implementation of Information Models in OPC UA and generation of code from them are discussed in previous work as well [43][44]. The OPC UA Java SDK has been developed to load and create the base address space from XML-files containing UaNodeSets. The SDK makes it possible to create the default Modbus Address Space by importing the Modbus UaNodeSet and then allowing the application to add or modify additional Nodes that use these generated Nodes during runtime. This means that the Nodes defined in section 5.1 will be imported to the Modbus Server. This enables creation of, for example, Modbus devices with the TypeDefinition ModbusMasterType. The Modbus Server will import the following required NodeSets to the OPC UA server: OPC UA base, OPC UA DI and the newly created OPC UA Modbus NodeSet.

Although the OPC UA server can import the NodeSets, developers will have a hard time using Nodes from that NodeSet as they are only generated at server startup and runtime. This issue is solved with Codegen in the Prosys OPC UA Java SDK that is created based on the research done by Laukkanen [43]. Codegen is used to generate interfaces and implementations of Java classes for type instantiation in the development process.

The Modbus communication of the application is created with an existing library called j2mod. The library provides a Java API for both creating and providing connections to Modbus devices. It handles requests, responses and creates exceptions on failure. The Modbus library polls Modbus slave devices cyclically and then writes the register values to the UA Variable Node. To keep the application code structural and modular the Modbus and OPC UA parts are still kept separated to keep the code readable and maintainable.

5.4 Application deployment

To handle the applications runtime environment, the application implements a Service Manager for handling service interactions on different platforms. As the design decisions in section 4.1 states that the application should be able to run headless as a service, the Service Manager keeps track of which operating system is used and whether the application is run as a service or as an application. It also handles starting and stopping of the service.

Because the application uses Java FX, it sets a limitation that a graphical user interface (GUI) is required for it to run. On Windows based computers this is not an issue as Windows always ship with a GUI and even when an application is run as a Windows service, a GUI will still be used. The Windows service manager runs applications on the system session, which is similar to a user session but its GUI

will never be visible to the user. Linux servers does not include a GUI at all so here the Java FX restriction becomes a problem. To bypass this issue a virtual display called X Virtual Frame Buffer needs to be used. It simulates a display in the background and thus allowing Java FX to run. Both the Debian and RPM packages of the application will therefore require installation of some dependencies alongside the main package for the application to be able to run correctly. As the supported Linux systems can use either SysV or Systemd as their init system, support for both is necessary. The installation package detects which init system is used and installs the corresponding service script.

5.5 User interface

The GUI of the Modbus Server will inherit the basic layout from the Simulation Server. The Modbus functionalities are designed to match the layout of the Simulation Server. The majority of the OPC UA settings will be hidden by default as they generally should not require configuration. Advanced users will still be offered the possibility to configure the OPC UA parameters if necessary by enabling an expert mode that will reveal additional configuration tabs. The user interface is created with the intention to minimize the possibility of making mistakes during configuration of the Modbus Server. This is done by trying to minimize the amount of interactions necessary to configure a Modbus device and is achieved by automatically configuring as much as possible, preventing faulty configuration parameters and by always confirming destructive actions.

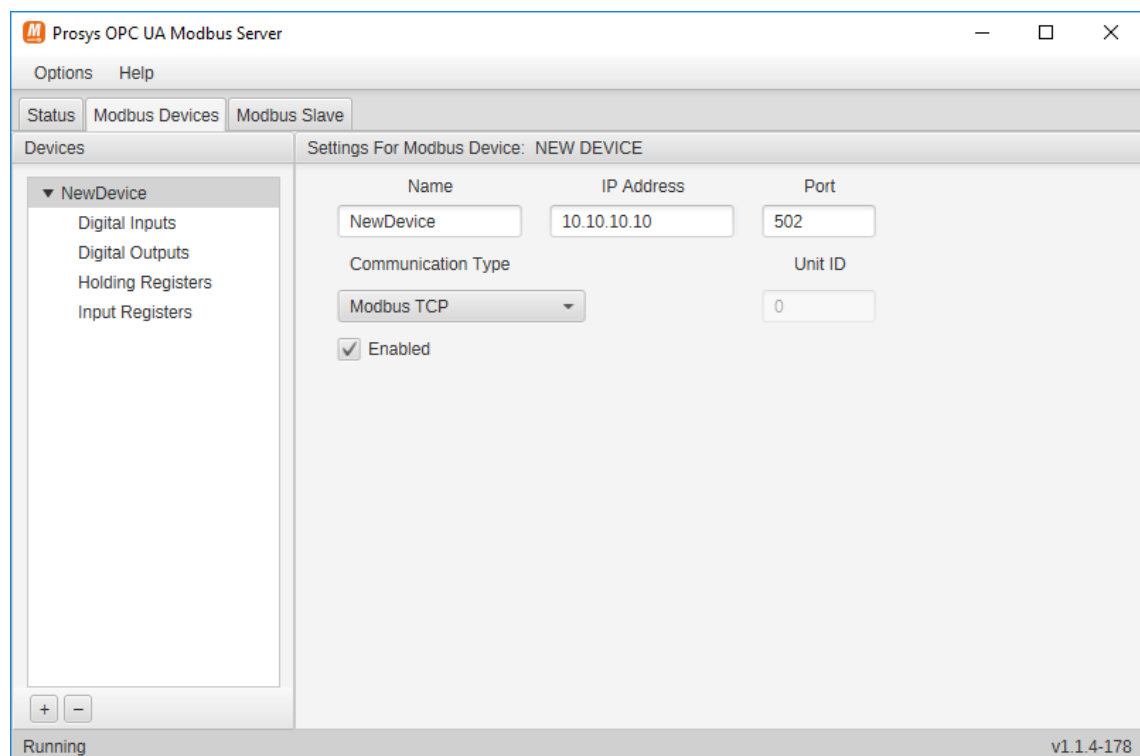


Figure 50: OPC UA Modbus Server Modbus Devices tab.

The Modbus master device parameters are configured on the Modbus Device configuration tab seen in figure 50. The created devices are listed in a tree view on the left side of the tab to be available quickly. The connection configuration of a device can be modified by selecting the device in the tree view. The buttons below the tree view allows the user to add or remove devices. IP addresses of the devices are validated and only IPv4 addresses are accepted. The port number can be set to any valid port number even though 502 is the standard, in case some device would implement a non-standard port. The communication type lists the supported communication types and as section 4.1 states, the Modbus Server only covers communication over TCP/IP. The unit id is only applicable when a device uses Modbus over TCP as then it will be needed to identify Modbus devices behind a gateway. If a parameter is invalid, a text message will inform the user about the issue so that it can be corrected.

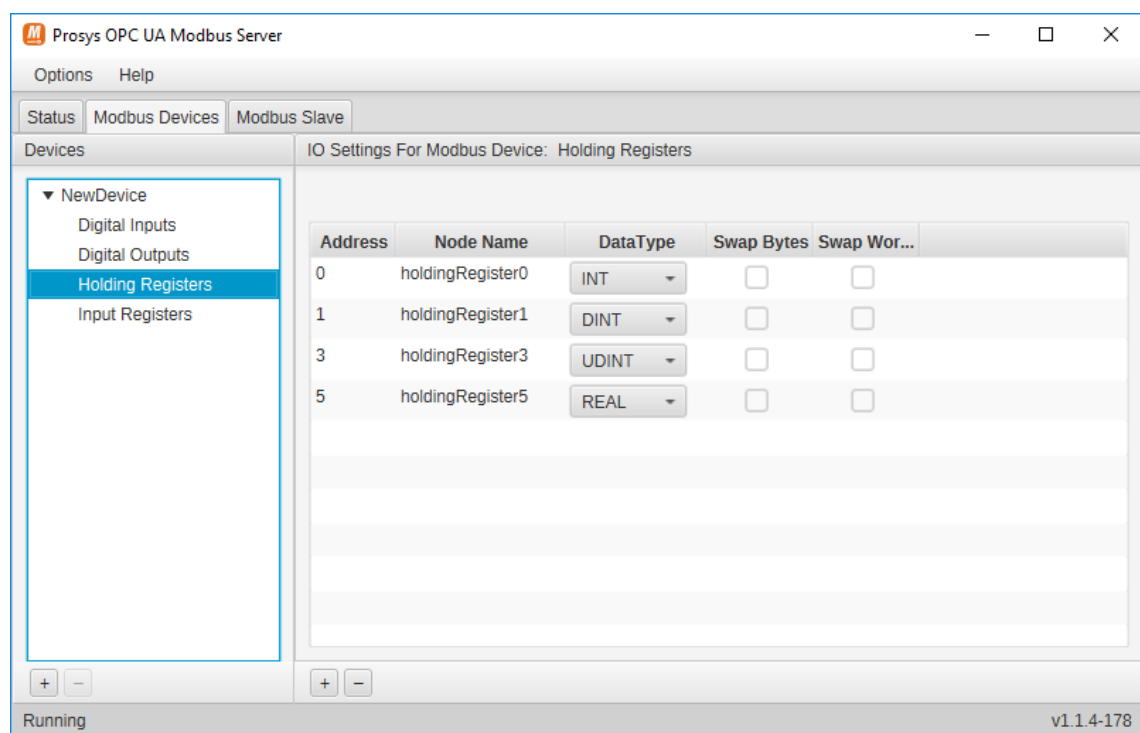


Figure 51: OPC UA Modbus Server Modbus register configuration view.

By expanding a device in the tree view, the four data tables are exposed. Selecting one of them changes the view on the Modbus Device tab so that the selected data table can be configured. Figure 51 shows the Holding registers table configuration with four configured Modbus variables. Each of the variables have a slightly different configuration. Variables can be added and removed from the table by clicking on the respective button below the table. The address of a Modbus variable can be set as long as it is within the Modbus addressing range and does not overlap another register. The variables can also be named freely but will always default to the table type and address, like holdingRegister0. The data type is chosen from a dropdown list containing the types listed in table 10. This is also restricted so that changing from a

16-bit DataType to a 32-bit DataType is only allowed if the change won't overlap an existing variable. The swap checkboxes are only enabled if the corresponding usage is possible. Let's say that the value is a 16-bit integer, the *swap words* checkbox is then disabled as word swapping is not possible with values that are only one word long.

The Modbus slave tab uses the same layout and configurations as the Modbus devices tab but as the usage of a Modbus slave is a bit different it is configured separately on its own configuration tab. The Modbus Server only allows creation of one Modbus slave as it should only be able to use the standard Modbus port 502 and that one slave will bind to that port exclusively [11].

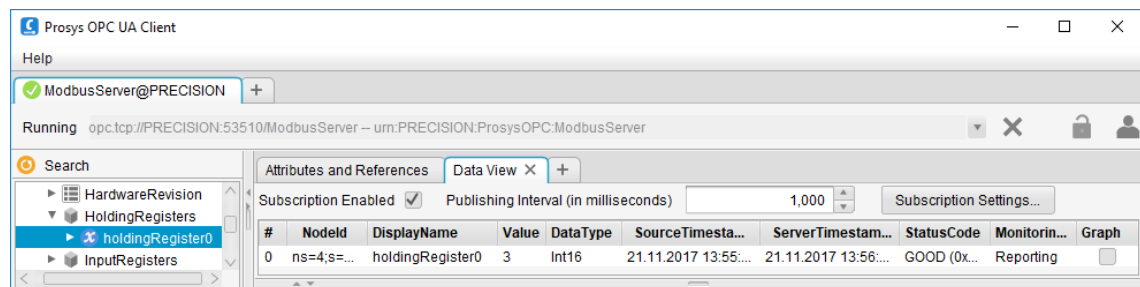


Figure 52: OPC UA Modbus Server value in OPC UA Client.

The OPC UA Modbus Server can be tested using the same methods that the evaluated applications in section 3.2 used. Adding a Modbus device to the Modbus Server is quicker than any of the evaluated applications as it only requires one click and setting the IP address. A register is easily and very quickly added as well, even if some of the other applications are basically as quick.

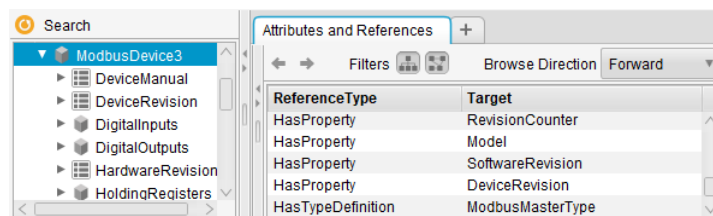


Figure 53: OPC UA Modbus Server device TypeDefinition.

By connecting an OPC UA client to the Modbus Server, variable values can be read. Figure 52 shows that the holdingRegister0 variable has a value of three and the data type Int16 which match its configuration in figure 51. By adding another two devices to the OPC UA Modbus Server and creating some Modbus variables in them, the modelled example Address Space in figure 43 matches the Modbus Server Address Space in figure 54. The Address Space is further examined by selecting a ModbusDevice and checking its TypeDefinition. Figure 53 shows that the device has the TypeDefinition ModbusMasterType that was created earlier in the Modbus Information Model and imported to the OPC UA server as an UaNodeSet.

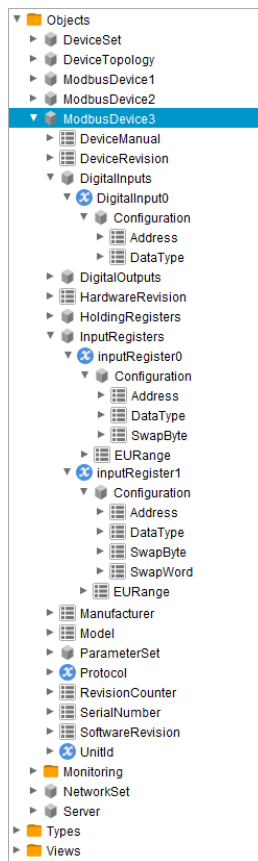


Figure 54: OPC UA Modbus Server example Address Space.

Continuing to one of the Modbus variable blocks like the holding register block, the selected holding registers Node in figure 55 is seen to have the TypeDefinition `ModbusHoldingRegisterBlockType`, that was defined in the Modbus UaNodeSet as well. The same figure also shows that the Modbus Variables are defined as components of the HoldingRegisters block.

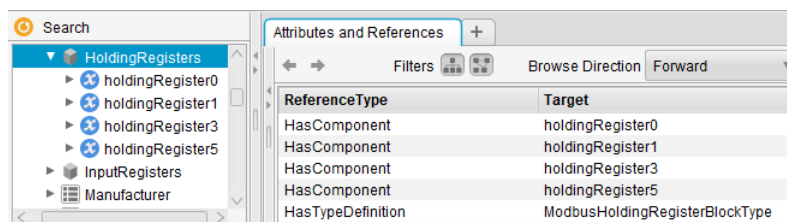


Figure 55: OPC UA Modbus Server IO-block TypeDefinition.

In section 4.2 it is stated that the Modbus variables are either analog or discrete items. The left side of figure 56 shows the attributes and references of a holding register that provide an analog variable and it can be seen that it has the Type-Definition `AnalogItemType`. As the variable can't have two type definitions it is given a `HasModbusRegisterConfiguration` reference to its configuration Node, in

addition to the HasTypeDefinition reference. To the right in figure 56 the configuration Node of the Modbus variable is examined. The configuration has the type definition ModbusRegisterConfigurationType and the properties Address, DataType and SwapBytes.

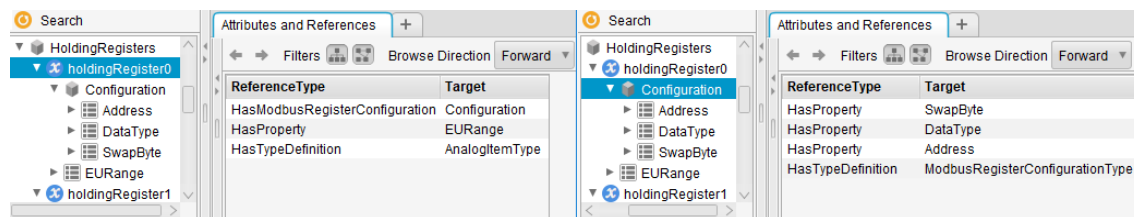


Figure 56: OPC UA Modbus Server Variable TypeDefinition.

The Modbus variable configuration parameters are particularly useful when using custom names for the variables like, for example, if holding register 10 would be named *Motor1.CurrentSpeed* it would be quite hard to tell anything about its Modbus configuration. So when the *Motor1.CurrentSpeeds* configuration then has the Address set to 10, the DataType set to INT and SwapBytes set to false, its Modbus configuration can quickly be determined by looking at the configuration Node. The type definitions may be very useful in many cases, one example though is that the user can easily keep track of which connected devices are Modbus devices. By using a custom OPC UA client the user could, for example, list only the Nodes that have the type definition *ModbusMasterType* to only see those devices in Modbus Server, even if there are several other types of devices and applications connected as well.

6 Conclusion

The development of the OPC UA Modbus Server showed that surprisingly many hours was necessary to generate a usable Information Model, even if Modbus is a fairly simple protocol. Finding the right level of abstraction, from a users point of view, was determined to be the most time consuming task. The development of the Modbus Server followed the currently popular agile development method and, for example, the Information Model was created following an iterative process by setting requirements, creating the model and reviewing the results and then restarting that process over again until a satisfying Information Model was found.

Reviewing similar products showed to be a very valuable source of information while creating requirements for the application as well. The review gave a lot of insight into what kind of features needed to be implemented and how the usability should be designed in relation to these applications. Especially the usability review gave a really good starting point for the requirements. As none of the applications had been used before, an inexperienced users point of view was established quite easily. It set certain demands on how to make the application as usable as possible.

To answer the three objectives set in the beginning of the thesis. Firstly, *How should data be structured and abstracted in an OPC UA application.* This is a very hard task and there is no straight answer to how it should be done as every application is different. An iterative process is highly recommended as the review of a finished Information Model or application will reveal flaws. Necessary improvements are more likely detected this way as well.

Secondly, *Can an OPC UA application be designed to be easy to set up.* That is definitely possible. The key, in this case, is to make the application as unambiguous as possible. Users should not need to configure the application too much, otherwise they get confused about where to configure which settings. They should neither be allowed to set faulty parameters. The usability design of the OPC UA Modbus Server succeeded very well and, compared to other similar applications, it is simple to set up and use.

Lastly, *What benefits does OPC UA have over wrapping with HTTPS.* This project showed that OPC UA is very suitable for transferring Modbus data. The benefits of using OPC UA is clearly that it has the ability to provide metadata as well as giving several additional features that are implemented in the OPC UA SDK while wrapping with HTTPS only provides data encryption. As seen in this thesis the semantics provided by the OPC UA Modbus Server can give valuable extra information about the application by creating useful metadata about the configured devices. An OPC UA client can furthermore be implemented to use this metadata in several ways according to customer needs.

In general this thesis might be a good source of information for someone starting with Information Model design. It gives a general idea on the flow of the design process by first creating a graphical model and then turning that model into an actual Information Model that can be imported into an OPC UA server. It can be especially valuable if another protocol, similar to Modbus is supposed to be mapped to OPC UA.

6.1 Future work

As every aspect of this kind of an application can not be covered in the bounds of one thesis, there are several features in the OPC UA Modbus Server that still could be improved and added. Some of the additions would require large rework and some only minor but as OPC UA showed to be very suitable for transferring Modbus data, the application will most likely be improved in the future.

The mapping of the IEC 61131-3 values to OPC UA could have been defined directly with the OPC UA for IEC 61131-3 elementary data types, which defines the same data types chosen in table 11 and several more in addition to them. Support for the rest of these data types will most likely be implemented in the near future.

Settings for the connection to a Modbus device is minimal and could, for example, be extended to let the user choose the polling rate instead of it being fixed to one second. This would enable reduction of network load by slowing down the polling rate to, for instance, five seconds or allowing the user to read critical data more often, for example, every 300 ms. It would also be useful if the application would give an indication of whether a device is connected or disconnected from the Modbus Server. Currently the only way to determine the connection status is by looking at the log file. If such a feature would be added, it would also be good to add preview of the Modbus device values. These improvements would be of major help in debugging as the user could easily determine if there is a problem with Modbus or OPC UA.

When the user has created the configuration, the application needs to be restarted in order to activate the new configuration. This feature generates unnecessary downtime, as normally the user only wants to add a device or even just a single variable to the configuration. Due to the application architecture of the OPC UA Simulation Server that was used as a base for this application, the implementation of that feature would require a very large redesign of the application architecture and as such was decided to be implemented in a later version of the application.

Allowing users of the OPC UA Modbus Server to create copies of the devices would be a valuable feature as in many cases the user wants to connect several similar devices. The configuration would be a lot faster by copying one configured device and only changing the name and IP address of the new copied device, instead of creating a completely new device and adding all the same Modbus variables as those added to the previously created device.

References

- [1] Rostan, M. (2014). *Industrial Ethernet Technologies*. EtherCAT Technology Group, Nuremberg, Germany. URL: https://www.ethercat.org/download/documents/Industrial_Ethernet_Technologies.pdf (Read 24.08.2017)
- [2] Miyachi, T., Yamada, T. (2014). *Current issues and challenges on cyber security for industrial automation and control systems*. SICE Annual Conference, Hokkaido University, Sapporo, Japan.
- [3] Sauter, T., Lobashov, M. (2011). *How to Access Factory Floor Information Using Internet Technologies and Gateways*. IEEE Transactions on Industrial Informatics, Vol. 7, No. 4.
- [4] Nardone, R., Rodríguez, R., Marrone, S. (2016). *Formal Security Assessment of Modbus Protocol*. IEEE, The 11th International Conference for Internet Technology and Secured Transactions.
- [5] Modbus Organization. (2017). *About The Protocol*. URL: <http://modbus.org/faq.php> (Read Jun 15, 2017).
- [6] Modbus Organization. (2017). *About The Modbus Organization*. URL: <http://modbus.org/faq.php> (Read Jun 15, 2017).
- [7] Clarke, G., Reynders, D., Wright, E. (2004). *Practical Modern SCADA Protocols: DNP3, 60870.5 and Related Systems*. Burlington, MA: Elsevier.
- [8] Modbus Organization. (2012). *MODBUS Application Protocol Specification V1.1b3*.
- [9] Modbus Organization. (2006). *MODBUS over Serial Line Specification and Implementation Guide V1.02*.
- [10] Modbus Organization. (2006). *MODBUS Messaging on TCP/IP Implementation Guide V1.0b*.
- [11] Internet Assigned Numbers Authority. (2017). *Service Name and Transport Protocol Port Number Registry*. URL: <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt> (Read Jun 20, 2017).
- [12] Schneider-Electric. (2017). *Modbus Plus*. URL: <https://www.schneider-electric.us/en/product-range-presentation/576-modbus-plus/> (Read Feb 19, 2018).
- [13] Simply Modbus. (2017). *Enron Modbus*. URL: <http://www.simplymodbus.ca/Enron.htm> (Read Feb 19, 2018).
- [14] OPC Foundation (2017). *Unified Architecture*. URL: <https://opcfoundation.org/about/opc-technologies/opc-ua/> (Read Jul 6, 2017).

- [15] Hoppe, S. (2017). *There Is No Industrie 4.0 without OPC UA*. OPC Foundation. URL: <http://opccconnect.opcfoundation.org/2017/06/there-is-no-industrie-4-0-without-opc-ua/> (Read Jun 20, 2017).
- [16] Prosys OPC (2017). *About OPC UA*. Espoo: Prosys OPC Ltd. URL: <https://www.prosysopc.com/opc-ua/> (Read Jul 6, 2017).
- [17] Aro, J. (2017). *OPC Day Europe 2017 - OPC UA is the Industry 4.0 Communication*. Espoo: Prosys OPC. URL: <https://www.prosysopc.com/blog/opc-day-europe-2017/> (Read Jul 6, 2017).
- [18] Mahnke, W., Leitner, S-H., Damm, M. (2009). *OPC Unified Architecture*. Berlin, Heidelberg: Springer Berlin Heidelberg.
- [19] OPC Foundation. (2015). *OPC Unified Architecture Specification Part 1: Overview and Concepts, Release 1.03*.
- [20] OPC Foundation. (2015). *OPC Unified Architecture Specification Part 2: Security Model, Release 1.03*.
- [21] OPC Foundation. (2015). *OPC Unified Architecture Specification Part 3: Address Space Model, Release 1.03*.
- [22] Unified Automation GmbH. (2017). *OPC UA NodeId Concepts*. URL: http://documentation.unified-automation.com/uasdkhp/1.0.0/html/_12_ua_node_ids.html#UaNodeIdsConcept (Read Jul 12, 2017).
- [23] OPC Foundation. (2015). *OPC Unified Architecture Specification Part 5: Information Model, Release 1.03*.
- [24] OPC Foundation. (2013). *OPC Unified Architecture for Devices, Companion Specification Release 1.01*.
- [25] OPC Foundation. (2015). *OPC Unified Architecture Specification Part 6: Mappings, Release 1.03*.
- [26] OPC Foundation. (2015). *OPC Unified Architecture Specification Part 7: Profiles, Release 1.03*.
- [27] OPC Foundation. (2015). *OPC Unified Architecture Specification Part 4: Services, Release 1.03*.
- [28] Belden Inc. (2013). *Manufacturing IT: Separate the Industrial Network from the IT Network*. URL: <http://www.belden.com/blog/industrialEthernet/Manufacturing-IT-Separate-the-Industrial-Network-from-the-IT-Network.cfm> (Read. 16.10.2017)
- [29] Loundsbury, R. (2008). *Industrial Ethernet on the Plant Floor: A Planning and Installation Guide*. Research Triangle Park: The Instrumentation, Systems, and Automation Society.

- [30] Microsoft. (2005). *Protocols and ports*. URL: <https://technet.microsoft.com/en-us/library/cc161377.aspx> (Read 30.10.2017).
- [31] Kepware. (2017). *KEPServerEX - Product Overview*. URL: <https://www.kepware.com/en-us/products/kepserverex/> (Read Oct 17, 2017).
- [32] Cogent. (2017). *Cogent DataHub*. URL: <https://cogentdatahub.com/products/cogent-datahub/> (Read Oct 19, 2017).
- [33] Cogent. (2017). *Modbus OPC Server*. URL: <https://cogentdatahub.com/products/datahub-modbus-opc-server/> (Read Oct 19, 2017).
- [34] Rock, A., Bouse, V. (2015). *The future is already here - Easy linking of production world and IT world through OPC UA*. Softing Industrial Automation GmbH, Haar, Germany.
- [35] Softing. (2017). *dataFEED OPC Suite*. Softing Industrial Automation GmbH, Haar, Germany. URL: <https://industrial.softing.com/en/products/software/opc-suite-servers-middleware/the-all-in-one-solution-for-opc-communication/datafeed-opc-suite.html> (Read 23.10.2017).
- [36] CommServer (2016). *CommServer UA*. URL: <http://www.commsvr.com/Products/OPCUA/CommServerUA.aspx> (Read 24.10.2017).
- [37] Duty, K. (n.d.). *3 Reasons Linux Is Preferred for Control Systems*. Inductive Automation, California, USA. URL: <https://www.automation.com/library/articles-white-papers/opc-articles-and-white-papers/3-reasons-linux-is-preferred-for-control-systems> (Read 01.11.2017).
- [38] OPC Foundation. (2015). *OPC Unified Architecture Specification Part 8: Data Access, Release 1.03*.
- [39] John, K-H., Tiegelkamp, M. (2010). *IEC 61131-3: Programming Industrial Automation Systems, Second Edition*. Berlin, Heidelberg: Springer Berlin Heidelberg.
- [40] PLCopen and OPC Foundation. (2010). *OPC UA Information Model for IEC 61131-3 - Release 1.00*.
- [41] Unified Automation GmbH. (n.d.). *UaModeler "Turns Design into Code"*. URL: <https://www.unified-automation.com/products/development-tools/uamodeler.html> (Read 02.11.2017).
- [42] Boström, B. (2014). *JavaFX based OPC UA Simulation Server*. Master's thesis, Aalto University.
- [43] Laukkanen, E. (2013). *Java source code generation from OPC UA information models*. Master's thesis, Aalto University.

- [44] Palonen, O. (2010). *Object-oriented implementation of OPC UA Information Models in Java*. Master's thesis, Aalto University.

A Modbus Information Model

The complete Modbus Information Model extends the Devices Information Model that itself extends the Base Information Model and its built in Data Access extension Information Model.

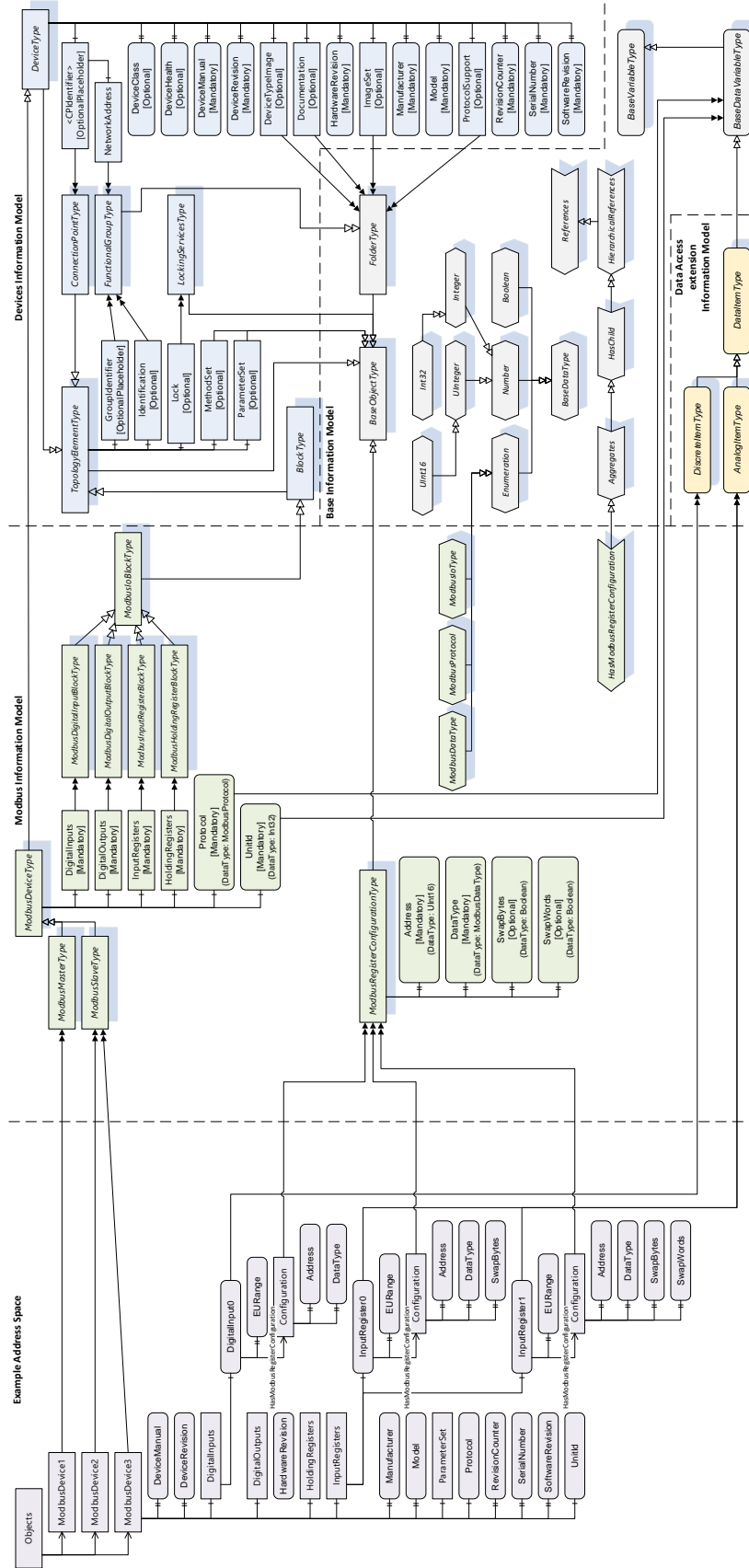


Figure A1: OPC UA Modbus Information Model

B Modbus NodeSet

The complete Modbus UaNodeSet created from the Modbus Information Model with Unified Automation UaModeler.

Listing 2: Modbus UaNodeSet

```
<UaNodeSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:uax="http://opcfoundation.org/UA/2008/02/Types.xsd"
  xmlns="http://opcfoundation.org/UA/2011/03/UaNodeSet.xsd"
  xmlns:s1="http://prosysopc.com/UA/Modbus/Types.xsd"
  xmlns:s2="http://opcfoundation.org/UA/DI/Types.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <NamespaceUris>
    <Uri>http://prosysopc.com/UA/Modbus/</Uri>
    <Uri>http://opcfoundation.org/UA/DI/</Uri>
  </NamespaceUris>
  <Aliases>
    <Alias Alias="Boolean">i=1</Alias>
    <Alias Alias="UInt16">i=5</Alias>
    <Alias Alias="Int32">i=6</Alias>
    <Alias Alias="LocalizedText">i=21</Alias>
    <Alias Alias="HasModellingRule">i=37</Alias>
    <Alias Alias="HasTypeDefinition">i=40</Alias>
    <Alias Alias="HasSubtype">i=45</Alias>
    <Alias Alias="HasProperty">i=46</Alias>
    <Alias Alias="HasComponent">i=47</Alias>
    <Alias Alias="ModbusDataType">ns=1;i=3001</Alias>
    <Alias Alias="ModbusProtocol">ns=1;i=3003</Alias>
  </Aliases>
  <Extensions>
    <Extension>
      <ModelInfo Tool="UaModeler" Hash="5c+oiEaVPismSBOUce+I+A==" Version="1.4.3" />
    </Extension>
  </Extensions>
  <UaDataType NodeId="ns=1;i=3001" BrowseName="1:ModbusDataType">
    <DisplayName>ModbusDataType</DisplayName>
    <References>
      <Reference ReferenceType="HasProperty">ns=1;i=6003</Reference>
      <Reference ReferenceType="HasSubtype" IsForward="false">i=29</Reference>
    </References>
    <Definition Name="1:ModbusDataType">
      <Field Name="BIT" Value="0"/>
      <Field Name="INT" Value="1"/>
      <Field Name="UINT" Value="2"/>
      <Field Name="DINT" Value="3"/>
      <Field Name="UDINT" Value="4"/>
      <Field Name="REAL" Value="5"/>
    </Definition>
  </UaDataType>
  <UaVariable DataType="LocalizedText" ParentNodeId="ns=1;i=3001" ValueRank="1"
    NodeId="ns=1;i=6003" ArrayDimensions="4" BrowseName="EnumStrings">
    <DisplayName>EnumStrings</DisplayName>
    <References>
      <Reference ReferenceType="HasProperty" IsForward="false">ns=1;i=3001</Reference>
      <Reference ReferenceType="HasModellingRule">i=78</Reference>
      <Reference ReferenceType="HasTypeDefinition">i=68</Reference>
    </References>
    <Value>
      <uax:ListOfLocalizedText>
        <uax:LocalizedText>
          <uax:Text>BIT</uax:Text>
        </uax:LocalizedText>
        <uax:LocalizedText>
          <uax:Text>INT</uax:Text>
        </uax:LocalizedText>
        <uax:LocalizedText>
          <uax:Text>UINT</uax:Text>
        </uax:LocalizedText>
        <uax:LocalizedText>
          <uax:Text>DINT</uax:Text>
        </uax:LocalizedText>
        <uax:LocalizedText>
          <uax:Text>UDINT</uax:Text>
        </uax:LocalizedText>
        <uax:LocalizedText>
          <uax:Text>REAL</uax:Text>
        </uax:LocalizedText>
      </uax:ListOfLocalizedText>
    </Value>
  </UaVariable>
  <UaDataType NodeId="ns=1;i=3002" BrowseName="1:ModbusIoType">
    <DisplayName>ModbusIoType</DisplayName>
    <References>
      <Reference ReferenceType="HasProperty">ns=1;i=6007</Reference>
      <Reference ReferenceType="HasSubtype" IsForward="false">i=29</Reference>
    </References>
    <Definition Name="1:ModbusIoType">
      <Field Name="DI" Value="0"/>
    </Definition>
  </UaDataType>
```



```

    <Field Name="DO" Value="1"/>
    <Field Name="IR" Value="2"/>
    <Field Name="HR" Value="3"/>
  </Definition>
</UADataType>
<UAVariable DataType="LocalizedText" ParentNodeId="ns=1;i=3002" ValueRank="1"
  NodeId="ns=1;i=6007" ArrayDimensions="4" BrowseName="EnumStrings">
  <DisplayName>EnumStrings</DisplayName>
  <References>
    <Reference ReferenceType="HasProperty" IsForward="false">ns=1;i=3002</Reference>
    <Reference ReferenceType="HasModellingRule">i=78</Reference>
    <Reference ReferenceType="HasTypeDefinition">i=68</Reference>
  </References>
  <Value>
    <uax:ListOfLocalizedText>
      <uax:LocalizedText>
        <uax:Text>DI</uax:Text>
      </uax:LocalizedText>
      <uax:LocalizedText>
        <uax:Text>DO</uax:Text>
      </uax:LocalizedText>
      <uax:LocalizedText>
        <uax:Text>IR</uax:Text>
      </uax:LocalizedText>
      <uax:LocalizedText>
        <uax:Text>HR</uax:Text>
      </uax:LocalizedText>
    </uax:ListOfLocalizedText>
  </Value>
</UAVariable>
<UADataType NodeId="ns=1;i=3003" BrowseName="1:ModbusProtocol">
  <DisplayName>ModbusProtocol</DisplayName>
  <References>
    <Reference ReferenceType="HasProperty">ns=1;i=6006</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">i=29</Reference>
  </References>
  <Definition Name="1:ModbusProtocol">
    <Field Name="TCP" Value="0"/>
    <Field Name="RTUTCP" Value="1"/>
    <Field Name="RTU" Value="2"/>
    <Field Name="ASCII" Value="3"/>
  </Definition>
</UADataType>
<UAVariable DataType="LocalizedText" ParentNodeId="ns=1;i=3003" ValueRank="1"
  NodeId="ns=1;i=6006" ArrayDimensions="4" BrowseName="EnumStrings">
  <DisplayName>EnumStrings</DisplayName>
  <References>
    <Reference ReferenceType="HasProperty" IsForward="false">ns=1;i=3003</Reference>
    <Reference ReferenceType="HasModellingRule">i=78</Reference>
    <Reference ReferenceType="HasTypeDefinition">i=68</Reference>
  </References>
  <Value>
    <uax:ListOfLocalizedText>
      <uax:LocalizedText>
        <uax:Text>TCP</uax:Text>
      </uax:LocalizedText>
      <uax:LocalizedText>
        <uax:Text>RTUTCP</uax:Text>
      </uax:LocalizedText>
      <uax:LocalizedText>
        <uax:Text>RTU</uax:Text>
      </uax:LocalizedText>
      <uax:LocalizedText>
        <uax:Text>ASCII</uax:Text>
      </uax:LocalizedText>
    </uax:ListOfLocalizedText>
  </Value>
</UAVariable>
<UAReferenceType NodeId="ns=1;i=4001" BrowseName="1:HasModbusRegisterConfiguration">
  <DisplayName>HasModbusRegisterConfiguration</DisplayName>
  <References>
    <Reference ReferenceType="HasSubtype" IsForward="false">i=44</Reference>
  </References>
  <InverseName Locale="2">ModbusRegisterConfigurationOf</InverseName>
</UAReferenceType>
<UAObjectType NodeId="ns=1;i=1002" BrowseName="1:ModbusRegisterConfigurationType">
  <DisplayName>ModbusRegisterConfigurationType</DisplayName>
  <References>
    <Reference ReferenceType="HasProperty">ns=1;i=6001</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6002</Reference>
    <Reference ReferenceType="HasSubtype" IsForward="false">i=58</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6004</Reference>
    <Reference ReferenceType="HasProperty">ns=1;i=6005</Reference>
  </References>
</UAObjectType>
<UAVariable DataType="UInt16" ParentNodeId="ns=1;i=1002"
  NodeId="ns=1;i=6001" BrowseName="1:Address">
  <DisplayName>Address</DisplayName>
  <References>
    <Reference ReferenceType="HasProperty" IsForward="false">ns=1;i=1002</Reference>
    <Reference ReferenceType="HasModellingRule">i=78</Reference>
    <Reference ReferenceType="HasTypeDefinition">i=68</Reference>
  </References>

```

```

    </References>
  </UAVariable>
  <UAVariable DataType='ModbusDataType' ParentNodeId='ns=1;i=1002'
    NodeId='ns=1;i=6002' BrowseName='1:DataType'>
    <DisplayName>DataType</DisplayName>
    <References>
      <Reference ReferenceType='HasProperty' IsForward='false'>ns=1;i=1002</Reference>
      <Reference ReferenceType='HasModellingRule'>i=78</Reference>
      <Reference ReferenceType='HasTypeDefinition'>i=68</Reference>
    </References>
    <Value>
      <uax: Int32>0</uax: Int32>
    </Value>
  </UAVariable>
  <UAVariable DataType='Boolean' ParentNodeId='ns=1;i=1002'
    NodeId='ns=1;i=6004' BrowseName='1:SwapByte'>
    <DisplayName>SwapByte</DisplayName>
    <References>
      <Reference ReferenceType='HasModellingRule'>i=80</Reference>
      <Reference ReferenceType='HasTypeDefinition'>i=68</Reference>
      <Reference ReferenceType='HasProperty' IsForward='false'>ns=1;i=1002</Reference>
    </References>
    <Value>
      <uax: Boolean>false</uax: Boolean>
    </Value>
  </UAVariable>
  <UAVariable DataType='Boolean' ParentNodeId='ns=1;i=1002'
    NodeId='ns=1;i=6005' BrowseName='1:SwapWord'>
    <DisplayName>SwapWord</DisplayName>
    <References>
      <Reference ReferenceType='HasModellingRule'>i=80</Reference>
      <Reference ReferenceType='HasTypeDefinition'>i=68</Reference>
      <Reference ReferenceType='HasProperty' IsForward='false'>ns=1;i=1002</Reference>
    </References>
    <Value>
      <uax: Boolean>false</uax: Boolean>
    </Value>
  </UAVariable>
  <UAObjectType NodeId='ns=1;i=1003' BrowseName='1:ModbusIoBlockType'>
    <DisplayName>ModbusIoBlockType</DisplayName>
    <References>
      <Reference ReferenceType='HasSubtype' IsForward='false'>ns=2;i=1003</Reference>
    </References>
  </UAObjectType>
  <UAObjectType NodeId='ns=1;i=1006' BrowseName='1:ModbusDigitalInputBlockType'>
    <DisplayName>ModbusDigitalInputBlockType</DisplayName>
    <References>
      <Reference ReferenceType='HasSubtype' IsForward='false'>ns=1;i=1003</Reference>
    </References>
  </UAObjectType>
  <UAObjectType NodeId='ns=1;i=1007' BrowseName='1:ModbusDigitalOutputBlockType'>
    <DisplayName>ModbusDigitalOutputBlockType</DisplayName>
    <References>
      <Reference ReferenceType='HasSubtype' IsForward='false'>ns=1;i=1003</Reference>
    </References>
  </UAObjectType>
  <UAObjectType NodeId='ns=1;i=1009' BrowseName='1:ModbusHoldingRegisterBlockType'>
    <DisplayName>ModbusHoldingRegisterBlockType</DisplayName>
    <References>
      <Reference ReferenceType='HasSubtype' IsForward='false'>ns=1;i=1003</Reference>
    </References>
  </UAObjectType>
  <UAObjectType NodeId='ns=1;i=1008' BrowseName='1:ModbusInputRegisterBlockType'>
    <DisplayName>ModbusInputRegisterBlockType</DisplayName>
    <References>
      <Reference ReferenceType='HasSubtype' IsForward='false'>ns=1;i=1003</Reference>
    </References>
  </UAObjectType>
  <UAObjectType NodeId='ns=1;i=1001' BrowseName='1:ModbusDeviceType'>
    <DisplayName>ModbusDeviceType</DisplayName>
    <References>
      <Reference ReferenceType='HasComponent'>ns=1;i=5001</Reference>
      <Reference ReferenceType='HasComponent'>ns=1;i=5002</Reference>
      <Reference ReferenceType='HasComponent'>ns=1;i=5004</Reference>
      <Reference ReferenceType='HasComponent'>ns=1;i=5003</Reference>
      <Reference ReferenceType='HasSubtype' IsForward='false'>ns=2;i=1002</Reference>
      <Reference ReferenceType='HasComponent'>ns=1;i=6009</Reference>
      <Reference ReferenceType='HasComponent'>ns=1;i=6008</Reference>
    </References>
  </UAObjectType>
  <UAObject ParentNodeId='ns=1;i=1001' NodeId='ns=1;i=5001' BrowseName='1:DigitalInputs'>
    <DisplayName>DigitalInputs</DisplayName>
    <References>
      <Reference ReferenceType='HasComponent' IsForward='false'>ns=1;i=1001</Reference>
      <Reference ReferenceType='HasModellingRule'>i=78</Reference>
      <Reference ReferenceType='HasTypeDefinition'>ns=1;i=1006</Reference>
    </References>
  </UAObject>
  <UAObject ParentNodeId='ns=1;i=1001' NodeId='ns=1;i=5002' BrowseName='1:DigitalOutputs'>
    <DisplayName>DigitalOutputs</DisplayName>
    <References>
      <Reference ReferenceType='HasComponent' IsForward='false'>ns=1;i=1001</Reference>

```

```

        <Reference ReferenceType="HasModellingRule">i=78</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=1;i=1007</Reference>
    </References>
</UAObject>
<UAObject ParentNodeId="ns=1;i=1001" NodeId="ns=1;i=5004" BrowseName="1:HoldingRegisters">
    <DisplayName>HoldingRegisters</DisplayName>
    <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=1;i=1001</Reference>
        <Reference ReferenceType="HasModellingRule">i=78</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=1;i=1009</Reference>
    </References>
</UAObject>
<UAObject ParentNodeId="ns=1;i=1001" NodeId="ns=1;i=5003" BrowseName="1:InputRegisters">
    <DisplayName>InputRegisters</DisplayName>
    <References>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=1;i=1001</Reference>
        <Reference ReferenceType="HasModellingRule">i=78</Reference>
        <Reference ReferenceType="HasTypeDefinition">ns=1;i=1008</Reference>
    </References>
</UAObject>
<UAVariable DataType="ModbusProtocol" ParentNodeId="ns=1;i=1001" NodeId="ns=1;i=6009"
    BrowseName="1:Protocol" UserAccessLevel="3" AccessLevel="3">
    <DisplayName>Protocol</DisplayName>
    <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        <Reference ReferenceType="HasModellingRule">i=78</Reference>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=1;i=1001</Reference>
    </References>
</UAVariable>
<UAVariable DataType="Int32" ParentNodeId="ns=1;i=1001" NodeId="ns=1;i=6008"
    BrowseName="1:UnitId" UserAccessLevel="3" AccessLevel="3">
    <DisplayName>UnitId</DisplayName>
    <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        <Reference ReferenceType="HasModellingRule">i=78</Reference>
        <Reference ReferenceType="HasComponent" IsForward="false">ns=1;i=1001</Reference>
    </References>
</UAVariable>
<UAObjectType NodeId="ns=1;i=1005" BrowseName="1:ModbusMasterType">
    <DisplayName>ModbusMasterType</DisplayName>
    <References>
        <Reference ReferenceType="HasSubtype" IsForward="false">ns=1;i=1001</Reference>
    </References>
</UAObjectType>
<UAObjectType NodeId="ns=1;i=1004" BrowseName="1:ModbusSlaveType">
    <DisplayName>ModbusSlaveType</DisplayName>
    <References>
        <Reference ReferenceType="HasSubtype" IsForward="false">ns=1;i=1001</Reference>
    </References>
</UAObjectType>
</UANodeSet>

```