Høgskulen på Vestlandet

Bacheloroppgave

ELE350

Predefinert informasjon

Startdato:	08-05-2023 09:00 CEST	**Termin:**	2023 VÅR
Sluttdato:	22-05-2023 14:00 CEST	**Vurderingsform:**	Norsk 6-trinns skala (A-F)
Eksamensform:	Bacheloroppgave		
Flowkode:	203 ELE350 1 O 2023 VÅR		
Intern sensor:	Adis Hodzic		

Deltaker

Navn:	Fredrik Fauske Skavlem
Kandidatnr.:	292
HVL-id:	591464@hvl.no

Informasjon fra deltaker

Egenerklæring *:	Ja
Inneholder besvarelsen konfidensielt materiale?:	Ja
Jeg bekrefter at jeg har registrert oppgavetittelen på norsk og engelsk i StudentWeb og vet at denne vil stå på vitnemålet mitt *:	Ja

Gruppe

Gruppenavn:	BO23EB-11 Unified Namespace
Gruppenummer:	18
Andre medlemmer i gruppen:	Jokubas Morsund, Mathias Normann Knutsen

Jeg godkjenner avtalen om publisering av bacheloroppgaven min *

Ja

Er bacheloroppgaven skrevet som del av et større forskningsprosjekt ved HVL? *

Nei

Western Norway
University of
Applied Sciences

BACHELOR THESIS:

BO23EB-11 Unified Namespace

<Mathias Normann Knutsen>
<Fredrik Fauske Skavlem>
<Jokubas Morsund>

19. May. 2023

Document Control

Report title BO23EB-11 Unified Namespace	*Date/Version* 19. May. 2023/0.10 *Report number:* B023EB-11
Author(s): Mathias Normann Knutsen Fredrik Fauske Skavlem Jokubas Morsund	*Course:* AUTB20 *Number of pages including appendixes* 150
Supervisor at Western Norway University of Applied Sciences Adis Hodzic	*Security classification:* Open
Comments: We, the authors, allow publishing of the report. Source code will not be publicly published due to legal conflicts.	

Contracting entity: Goodtech	*Contracting entity's reference:* None
Contact(s) at contracting entity, including contact information: *Svein Borlaug*	

Revision	**Date**	**Status**	**Performed by**
0.10	19.05.23	First issue	MNK, FFS, JM

Preface

This thesis will be submitted to fulfill the Bachelor's degree in Automation with Robotics and will conclude our time at the Western University of Applied Science. The work in the thesis was carried out during the spring semester of 2023 under the supervision of Adis Hodzic.

The thesis has been inspired by Goodtech, which suggested the problem of system design and architecture based on experience from current projects and customer interest. Goodtech is a leading company in the field of IT and automation solutions. Founded in 1913, they have delivered automation systems for over 100 years and have been part of the innovation that has taken place over the last century. Goodtech focuses on providing intelligent solutions to customers that use the latest technology and make their business more efficient. Over the past few years, they have seen the need to re-think and redesign new system architectures due to current data collection and acquisition trends.

Much of the work on this thesis has been to combine already existing technology into a larger system. A substantial part of our time has therefore been spent acquiring the necessary insight to deploy and integrate different devices into a unified system architecture.

Summary

The industry constantly evolves, leveraging newly developed technologies to maximize profits while minimizing costs and environmental impacts. One recent trend is the collection of vast amounts of data, which can be used to optimize processes by identifying potential areas for improvement or enhancing the ability to rapidly adapt to emerging trends facilitated by real-time insights across the entire enterprise. Central to this trend is data sharing involving cross-vendors and protocol interoperability, essential for seamless information exchange between data sources and the systems that utilize the information.

A prominent example of data exploitation is AI which has achieved impressive results, showcasing the value of leveraging historical data to gain a competitive advantage over like-sided companies. In the past, data used to be isolated within the devices that produced it. However, companies now recognize the value of collecting and storing big data in data centers, often referred to as data lakes. Nevertheless, a challenge remains as the data within these centers is often without labels or context, limiting its usefulness for analytical tools that are increasingly prevalent in the IT domain. This motivates the need to model information and the extension of automation systems to be directly connected to IT networks, including the cloud, to fully capitalize on these benefits.

In this thesis, an architecture based on the Message Queuing Telemetry Transport (MQTT) protocol is proposed and evaluated. A strategy for implementing a software-based system for routing information, similar to how hardware routers function, is outlined. This approach challenges the traditional principle of isolation and segmentation commonly employed in automation systems to enhance cybersecurity. Consequently, the open architecture enabling seamless information flow across the entire enterprise must be balanced against security concerns. A proof of concept is realized to demonstrate the feasibility of the proposed system. Finally, the thesis concludes by summarizing key findings and providing recommendations on how a unified architecture can be implemented in a newly developed system or alongside an existing system in operation.

Sammendrag

Industrien utvikler seg kontinuerlig og benytter seg av nylig utviklede teknologier for å maksimere profitt samtidig som man minimerer kostnader og miljøpåvirkning. En nylig trend er innsamling av enorme mengder data, som kan brukes til å optimalisere prosesser. Optimaliseringen gjøres ved å identifisere potensielle områder for forbedring og evnen til å tilpasse seg raskt til nye trender ved hjelp av innsikten dataen gir. Sentralt i denne trenden er deling av data mellom ulike leverandører og protokollinteroperabilitet. Dette er essensielt for sømløs utveksling av informasjon mellom datakilder og systemene som bruker informasjonen.

Et fremtredende eksempel på utnyttelse av data er kunstig intelligens (KI) som har oppnådd imponerende resultater og har demonstrert verdien av å utnytte historiske data for å oppnå en konkurransefordel over andre selskaper. Tidligere var data isolert innenfor enhetene som produserte dem, men nå begynner selskaper å se verdien av å samle inn og lagre store mengder data i datasentre, ofte referert til som "datalakes". Denne oppsamlede dataen lider ofte av en stor utfordring, nemlig mangel på kontekst og identifikasjonsattributter. Dette begrenser bruken av analytiske verktøy, som blir stadig mer utbredt innenfor IT-domene. Derfor er insentivet stort for å modellere informasjonen og utvide automasjonssystemer med forbindelser til IT nettverk, slik at data automatisk gis kontekst når de lagres, for å benytte seg av fordeler analytiske-verktøy og KI gir.

I denne avhandlingen vil en arkitektur basert på protokollen Message Queuing Telemetry Transport (MQTT) bli drøftet og evaluert. Avhandlingen vil gi en strategi for å implementere et programvarebasert system for sømløs informasjon flyt innad i en virksomhet. Dette systemet vil utfordre de tradisjonelle prinsippene om isolasjon og segmentering, som vanligvis brukes i automasjonssystemer for å forbedre cybersikkerhet. Derfor må den åpne arkitekturen som muliggjør sømløs informasjonsflyt på tvers av hele virksomheten, balanseres mot potensielle sikkerhetstrusler. Et konsept har blitt realisert for å demonstrere gjennomførbarheten av det foreslåtte systemet og eventuelle muligheter og problemstillinger realiseringen vil medbringe. Avhandlingen avsluttes ved å oppsummere sentrale funn og gi anbefalinger om hvordan en kan fase inn et slikt system, samtidig som eksisterende system er i drift.

1 Table of Content

Figure List

Abbreviations

AES	Advanced Encryption Standard
AI	Artificial Intelligence
EMI	Electromagnetic Interference
FB	Function Block
GND	Ground
GPL	General Public License
HMI	Human Machine Interface
IC	Integrated Circuit
IT	Information Technology
JTAG	Joint Test Action Group
MCU	Microcontroller Unit
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
NTP	Network Time Protocol
OOP	Object Oriented Programming
OPC UA	Open Platform Communications Unified Architecture
OS	Operating System
OSI	Open Systems Interconnection
OT	Operational Technology
PCB	Printed Circuit Board
PLC	Programmable Logic Controller
PoE	Power over Ethernet
PnP	Plug and Play
PSK	Pre-Shared Key
QoS	Quality of Service
SIG	Signal
SBC	Single Board Computer
SSH	Secure Socket Shell
SOA	Service Oriented Architecture
TCP	Transport Control Protocol
TLS	Transport Layer Security
UDP	User Diagram Protocol
UNS	Unified Namespace
VM	Virtual Machine
VPN	Virtual Private Network
X509	Standard for digital certificates

2 Introduction

2.1 Background and motivation

New technology and automation systems are constantly being adopted as the industry is evolving. Significant development steps are characterized as industrial revolutions. Currently, Industry 4.0 [7] is the fourth industrial revolution, which embraces the leading trends in the exchange of data, autonomous systems, artificial intelligence, and the Internet of Things (IoT) in manufacturing systems. There is an increased effort to optimize production by creating smart factories of interconnected devices that share data at high speed. Overall, industrial revolution 4.0 is characterized by digitalization, increased efficiency, productivity, and the ability to rapidly adapt to received inputs from the environment, like customer demand or vendor delivery changes. It will likely have significant social and economic impacts on the industry's operations.

Digitalization [8] is a crucial aspect of the industrial revolution 4.0. It can be challenging to define digitalization. One way to describe it can be the process of converting information, processes, and systems into digital formats, making them easier to access and more efficient to manage. On the other hand, digital transformations are the processes of implementing digital technology that utilizes the increased amount of available data to automate and streamline various processes, such as data collection, analysis, and communication. Digitalization is transforming almost every sector of society, like healthcare, finance, education, and the manufacturing industry. The benefits are improved decision-making through data analytics and visualizations, improved accuracy, and increased efficiency by automating tasks previously done manually.

The use of data is central to making the industry more efficient. Decision-makers can use data to better understand and make informed decisions by providing a basis for analysis and comparison [9]. For example, sales data can provide trends and identify growth opportunities or areas which may be improved. Customer behavior and preference data can be used to improve products or services. Historical data from an automation line can help identify bottlenecks or potential waste of material and energy that can be optimized for better performance. One central tool, Machine learning (ML), can use historical data to predict machine failures [10], thereby implementing a maintenance schedule based on the state of the plant instead of periodic intervals reducing unnecessary downtime and cost.

Information Technology (IT) systems are computer systems used for data-centric computing. These systems are typically used in office environments and are designed to support a wide range of business operations, such as data management, communication, and decision-making. On the other hand, operational technology (OT) systems are used to monitor and control physical machines, processes, and electromechanical devices in industries like factories or oil and gas production. OT systems are often used to automate and optimize these industrial processes. They include sensors, actuators, control systems, and other devices used to collect data and control equipment in real-time. Today, IT and OT systems are often combined to facilitate exchange of data and enable more efficient and effective operation of the industrial process. However, they serve different purposes and are typically designed and managed separately [11, 12].

Plug and Play (PnP) is a term used to describe devices designed to be easily connected to a computer and configured for use without additional software installation or setup. A PnP device should be recognized by the system and made available for use as soon as it is connected. A prerequisite for PnP systems is that hardware and protocols are standardized. Devices used in IT systems have traditionally been more homogeneous than OT devices due to the implementation of standardization. An example of standardization is how devices seamlessly make themself known to others when added to a Wi-Fi or Bluetooth network without any intervention from the user. OT systems developed by vendors are often proprietary and have the implementation details hidden, making combining products from different suppliers more difficult [13]. Increased standardization will be necessary for OT systems to become PnP devices that can be added and removed without the complexity and at the increased cost compared to IT devices.

Typically, connections are established between devices that produce data (publishers) and those who want access to it (subscribers). Consequently, many point-to-point links are established, resulting in a spaghetti structure and a tightly coupled system, making it difficult to add or remove new devices without disturbing or breaking the existing communication structure. A better principle and a prerequisite for PnP functionality are to make data producers decoupled from the consumers. Decoupling is accomplished by a broker, described in detail later, functioning as an intermediate link through which all data passes. If information definitions are pre-defined and the message broker has the capability to temporarily store messages destined for subscribers, decoupling in time, location, and information formats can be achieved. This decoupling allows for more flexibility and scalability in distributed systems [14]. It makes it possible to connect and disconnect producers and consumers freely and in arbitrary order, with no requirement for direct links between devices.

One of the primary challenges to overcome is suppliers developing systems solely for their own product line, resulting in solutions that work well in isolation but do not have the opportunity for interoperability. Consequently, customers get locked to a specific supplier due to the financial cost of integration with non-supplier systems. Open System Interconnection (OSI) model is a framework to separate and standardize various functions in the transfer of information, enabling suppliers to develop at certain levels in the communication stack. Interoperability is guaranteed by defining interfaces between the layers. The model is divided into lower layers that describe the physical management of bits in the communication medium, the layers in the middle that handle addressing between devices, and higher layers that describe information encoding and structuring. A prerequisite for cross-vendor communication is that data arrives at the communicating parties, which is probably why the model's lower levels are almost standardized without many options to choose from. TCP/IP has become the de facto standard that most suppliers follow. Today’s challenges lie at the upper layers that describe how data is structured. A prerequisite for full interoperability without requiring extensive manual integration is that these layers are also standardized [15].

Understanding the context of the data is crucial for accurately interpreting and using it effectively. "Data context" refers to the state or environment in which data was collected or used. It includes additional information surrounding the data, such as the location, the time when it originated, reason for why it was collected, as well as any relevant background information[16]. For example, temperature and humidity measurements collected in a room will be more valuable if we know the circumstances surrounding the data. The context for this data can include information about the date and time of day when the data was collected, the room's location, and any relevant factors, such as the size of the room, the details about entries into the room, or the presence of any heat sources.

Computer programs must uniquely name variables and functions to address them in code modules. "*A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries*" [17]. Variables hold the data produced in the system or process under control. A namespace provides a way to guarantee globally unique names for identifiers across different vendor deliveries. This allows identifiers with the same name to co-exist without causing name clashes. The downside is that data often end up isolated inside the node that produced it, and it is often costly and complex to make it globally available. The concept of a Unified Namespace (UNS) creates an address space that combines multiple separate namespaces into a single one. A unified namespace is not a technology but a conceptual design principle enabling data addressing across the entire enterprise.

Increased amounts of data have provided new opportunities but have also caused challenges because of the increased amount of traffic between network devices that process and share the data. Traditionally, communication between devices has been done by pull and response requests. The device that will use the data had to request it frequently enough to ensure that changes in values are updated with a high enough frequency for the control task. Update times can be down to milli- or microseconds for processes with fast dynamics. A need to reconsider how and when devices communicate data across the network was needed. To minimize the traffic, several developers have therefore started to use event-based communication based on publish and subscribe messaging. The idea is that data should only be sent when it has changed. Subscribers will then subsequently receive the updated values. This way, network traffic is reduced by up to 90% [18]. However, there are downsides to the Subscriber/Publisher architecture. Systems that are consumers of data can no longer depend on periodic samples and must be modified. More important, if no updates are received, it must be because the values have not changed. Mechanisms to determine if devices have failed and are no longer online must be included in the system.

Making the systems available and dependent on networks also added additional challenges such as cyber security and robustness that must be considered. It is not uncommon for organizations to isolate certain systems or networks to improve their security. This practice is often referred to as air gaping, segmentation, or creating a "security island" [19]. Having isolated systems makes it more difficult for attackers to gain access to them. Unauthorized access and preventing malware from infecting or spreading across systems or networks is minimized by restricting or blocking all traffic into the network. Although it has improved the cyber security problem by isolating the systems, it is an obstacle to data sharing or utilizing cloud-based tools already developed. Cryptographic tools and technologies for securing and verifying information transfer have therefore been developed to overcome security concerns. It is essential to thoroughly understand the strengths and weaknesses of these technologies when exposing the entire enterprise data on the web. Cyber-attacks can have devastating economic and social consequences for businesses and must be upheld as a significant threat to any enterprise.

Not all systems can or should be dependent on data communication through the UNS. Safety or Real-time systems dependent on reliable and frequent sensor updates and actuator commands with microseconds requirements are unsuited for signal flow through a cloud broker. In the event of a communication failure, a safety system should ensure that the physical process always stays safe. These systems should be designed as fail-safe independent nodes, only receiving external setpoints. Alternatively, the broker can be situated on the local network. Still, deterministic requirements can be an issue for time-critical systems. On the other hand, analysis and visualization systems often do not have the same requirements for real-time updates as those that control and regulate the physical process. But requirements for reliability and timestamping can still be as important. For example, the real edge on a signal change can be decisive when a machine learning (ML) algorithm predicts the need for maintenance or estimates newly optimized parameters based on historical data. Correct timestamping is part of the data context that must accompany the data, even if the transfer is delayed.

2.2 Problem description

The concept of a Unified Namespace has been agreed upon in theory, but there seems to be a lack of experience from actual implementations. Practical experience is needed to advise customers better when they desire to digitalize their processes and lift the automation data to a UNS where it can be made available. Often, the customers' systems are in production, making it expensive and therefore not desirable for them to be stopped when making the necessary changes required to move them into the cloud. There is often a significant complexity and cost in modifying these systems.

In this thesis, we will try implementing a UNS on a cloud platform using a centralized data broker and publish and subscribe communication. We will attempt to connect different devices from different suppliers to determine PnP capabilities and ease of expanding or modifying a system that is in production. Our goal is to determine a strategy for how these systems can be gradually interconnected, preferably in a decoupled method to make them available in the cloud. Also, recommendations for designing systems not yet built to make them compatible with a UNS from the initial starting point will be explored in this thesis.

A requirement for operability is that hardware and protocols are agreed upon. The UNS's success will depend on suppliers and integrators moving toward a common hardware and software platform. Open-source technology has the advantage of being freely available to all. It is being developed and battle-tested by the entire online community. Consequently, guides and example codes are published on public forums, which helps reduce implementation costs. An example is Cryptography which is open-source. The fundamental principle of cryptographic algorithms are freely available to the public and have been extensively tested by the entire community for decades. Cryptographic security lies in the encryption keys and not in the implementation details of the algorithm. Goodtech desires that we strive to use open-source protocols and the well-agreed-upon interface of Ethernet and TCP/IP as a foundation for data communication.

An essential requirement for industrial systems is uptime and security. The system design must be robust so that errors and unforeseen events do not lead to unnecessary downtime or dangerous situations that should have been avoided. The system must presume a safe state in the event of a failure and return to normal operating condition after failure recovery. Before the concept of a UNS can be recommended to customers, reliability and robustness must be tested and documented. One of Goodtech's main priorities is for us to look into how we can test and document the system's robustness.

Goodtech is not recommending that a customer is involved as part of the thesis based on previous experience conducting student thesis in cooperation with customers. Customer involvement has previously led to delays in the project workflow, absolute deadlines, and less flexibility. It is more desirable for Goodtech to have more influence and control of the project direction. The thesis will therefore be conducted as a proof of concept. The scope will be flexible and allow us to freely decide during each stage on what technologies to include in the UNS prototype, also dependent on available time.

In summary, the thesis goals are:

- Identify potential solutions for integrating OT and IT systems with reliability, security, expandability, and decoupling in time and location as main priorities. Strive for open-source solutions where possible.

- Suggest a strategy for data context and organizing the address space of the UNS with generality and the ability to expand as bearing principles.

- Verify and document the reliability, security, and robustness on a realization of a proof-of-concept system.

2.3 The main idea of the solution

The strategy for the thesis UNS system will be to implement a system parallel to the one in operation where current and future data producers and consumers can be added over time. The intent is to grow the system while parts are online and under regular operation to determine the system's ability to expand without re-configuration or breaking already existing nodes. HiveMQ is a prevailing alternative for a data broker with an active community and educational materials readily available. It will be the technology on which we base the UNS in this thesis. Different solutions for broker implementation and the possibility of a hierarchical architecture of brokers under the canopy of the HiveMQ broker will be investigated as part of the thesis.

Common practice today is that the suppliers delivering automation systems at the edge are designing primarily with the unit control objective in mind and with fewer considerations for interoperability. A standard solution is to create a separate system connecting to each of the different units to centralize the factory state on a common user interface. It has usually led to many point-to-point connections, which make it almost impossible to change or modify the system without extensive re-configuration of existing devices. The design intent of the UNS is to make the system centralized around a broker. Implementing the broker will be part of defining the principles for communication, rules for data contexts, and information definitions in the Namespace. New nodes connecting to the UNS must adapt to be compatible with the existing configuration instead of being the ones that set the conditions.

One of our main design principles will be to explore and implement open-source technologies. But to gain better insight, we will attempt to develop our own gateways as bridges between legacy devices that run on the most common protocols in use today. These gateways will be implemented as software modules that run on clients connected to the same network as the legacy devices. We will strive to use standard libraries developed and released to the public by the companies responsible for maintaining the protocol or open-source libraries maintained and tested in cooperation with the public.

To demonstrate the simplicity of adding new devices to the system, we will develop our own IIoT device. It will be based on a microprocessor on an in-house designed circuit board. It will be developed in isolation without needing to know how the rest of the system is built. The device will implement the MQTT protocol and standardize information to be recognizable inside the UNS. This should demonstrate that any sensor from any supplier will be able to send data to the UNS and that consuming devices can obtain and utilize this data without significant integration work.

To test the system, we will use Ignition, a widely used Supervisory and Data Acquisition (SCADA) system in the industry, to verify the availability of data published to the Namespace. We will also design new data consumers whose sole aim is to verify the system's robustness by logging data that has been received and identifying whether something was lost during transit. Data distributed through the broker is essentially volatile data that is not stored. It will therefore be of interest to explore solutions for historians (databases) that can be connected as a subscriber to the UNS to store this data for long-term access.

Our end goal is to implement a complete system consisting of edge PLCs publishing process data and receiving external setpoints to a hierarchical cluster of local and cloud brokers. A dashboard and historian will be configured as subscribers to visualize real-time data and for long-term storage. We will attempt to add IIoT devices from various suppliers with different communication standards into the same infrastructure to prove cross-vendor interoperability, always with concerns for cyber security and reliability in mind. OPC UA and MQTT, the two most widely used protocols for industrial IoT [15], cannot be overlooked and will be central technologies in our system. As an edge system, we plan to connect the educational PLC station at HVL, attempting to use OPC UA translating gateways and the native MQTT protocol incorporated in the S7-1500 CPU for communication to the UNS.

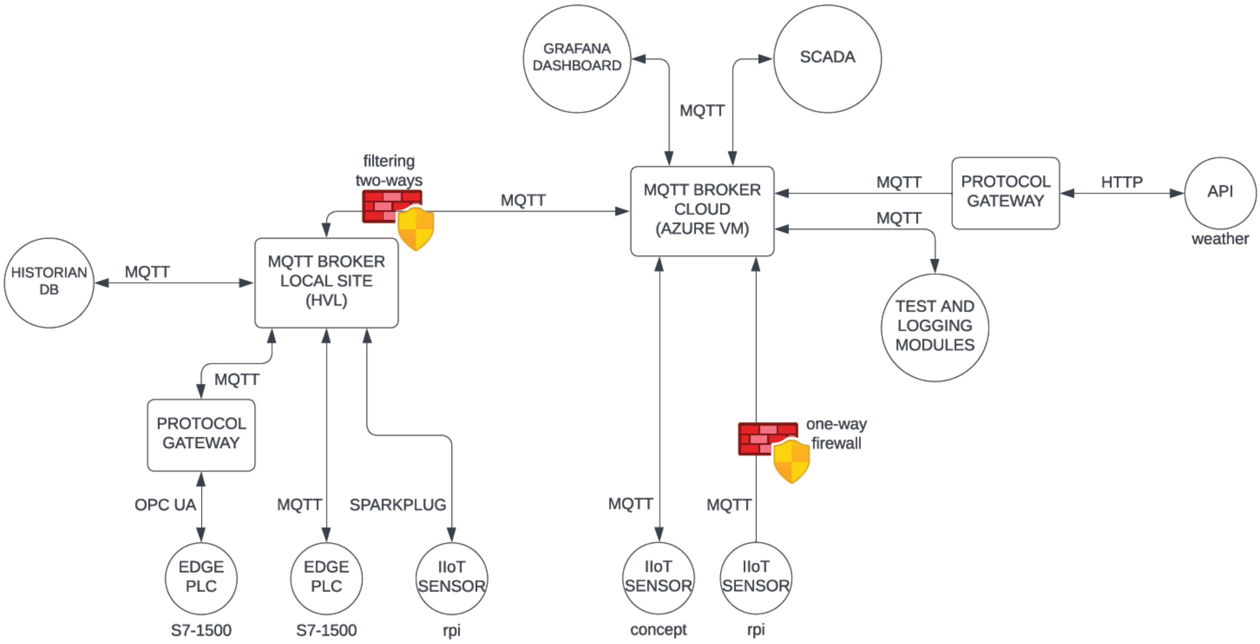


Figure 1 Diagram showing the initially planned bachelor thesis end result

2.4 Thesis Layout

The thesis is organized into seven parts and is comprised of the following sections:

Chapter 3 presents the relevant background theory of the related technologies and concepts that the Unified Namespace is built upon, including the most common protocols for device interoperability in use today, fundamentals of cyber security, and information modeling and standardization. Readers who are familiar with these concepts can skip the chapter without missing the general intent of the thesis.

In chapter 4, the realization of the Unified Namespace is described, including a complete system topology. Strategies and necessary technologies for connecting legacy systems are described in detail. Most of the implementation details have been abstracted from the main report to keep it concise but can be found in appendices for interested readers.

Further, Chapter 5 presents test procedures and results for verifying overall system performance, robustness and recovery.

Then, in Chapter 6, both the theoretical and practical experiences are discussed to determine recommendations for UNS planning, including the feasibility of implementing a unified namespace parallel to existing systems. Finally, Chapter 7 concludes the thesis.

3 Background theory

The reader is assumed to have a basic understanding of networking and related technologies to understand the work done in this thesis. Chapter 3 briefly describes these technologies as a way for the reader to understand the principal building blocks of the proposed architecture. It can be read in its entirety or used as a reference.

3.1 Network Communications

3.1.1 OSI model

The OSI model is a widely used reference model to explain the different functions needed for network communication. Each layer has protocols with a specific purpose and a defined interface to the layer below and above. The modularization of the communication stack enables developers to work on isolated network functions. It is also part of future-proofing the technology by allowing the exchange of individual protocols in the stack because of security issues or technology being deprecated.

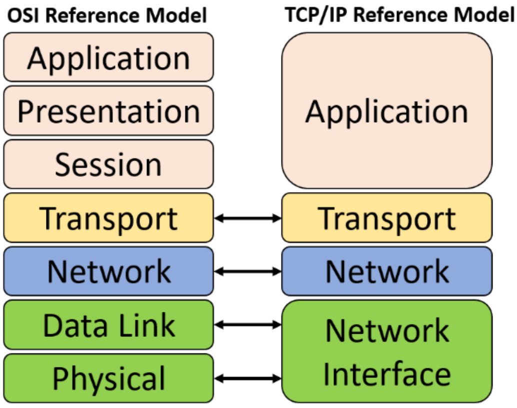


Figure 2 OSI compared to TCP/IP model

1. Physical layer handles the electrical transmission of bits on the wire or air.
2. Data Link handles the transmission and error detection/correction of data on the physical link.
3. Network layer handles logical addressing and routing between hosts on the network.
4. Transport layer handles end-to-end delivery of data, flow control, and error recovery.
5. Session layer establishes and manages sessions between hosts.
6. Presentation layer handles the encoding, compression, and encryption of data.
7. Application layer is the endpoint and utilizes the data transmitted over the network.

Both the OSI and TCP/IP models are widely used as a reference for teaching and understanding network communication.

3.1.2 TCP / IP model

The TCP/IP stack defines four layers that map to equal layers in the OSI model. The upper three levels of the stack have been combined into one, emphasizing that it is more focused on the flow of data between hosts, not the internal processing by the host. The lower levels in the models are almost identical and illustrate that this part of network communication has practically become standardized.

3.1.3 Ethernet

Ethernet is a physical and data link technology that can run on various media like copper or fiber. It is commonly used to connect hosts in Local Area Networks (LAN) or Wide Area Networks (WAN). The ethernet standard is continuously updated to support higher bit rates, increased number of nodes, and longer distances. Each data stream is divided into ethernet packets called frames with the sender and receiver MAC addresses and error detection codes. The standard has been the de facto technology for network transmission.

3.1.4 Transport layer protocol

The Transmission Control Protocol (TCP) and User Diagram Protocol (UDP) represent the transport layer. TCP is a non-deterministic and connection-oriented protocol that establishes and maintains point-to-point communication between hosts. It is a reliable protocol that ensures packet ordering, error detection, and re-transmission of lost packets. Although it is reliable, transmission is not deterministic, making it less suited for industrial networks with real-time dependencies. The protocol has significant overhead resulting in greater use of bandwidth. An advantage is that protocols above the Transport Layer often do not implement functionality to ensure reliable communication when the transmission is over TCP.

UDP is a non-deterministic and non-reliable protocol that does not verify or order packets at the receiver. It is comparatively faster, simpler, and more efficient than TCP. There is no overhead for establishing connections, error checking, or termination. The protocol is best suited for communication where bandwidth is most important, and re-transmission of lost packets is not as crucial. Examples of used cases are constant streaming of data like video or high-speed sensor updates.

3.1.5 Internet protocol

The transport layer protocols run on top of Internet Protocol (IP) which ensures that destination hosts are uniquely identified on the network. Each host has an IP address consisting of 32 bits for IPv4 and 128 bits for IPv6. The IP address contains a network part identifying the subnet and a host part identifying the device on the subnet. Hosts are commonly separated into subnets, identified by a sub mask, to reduce the amount of traffic and to enable unique addressing within the subnets. If the destination IP address is unknown to a device, it will forward it to the configured gateway that handles routing between subnets, including the internet. It is the responsibility of the IP protocol to maintain routing tables so that paths to remote hosts can be identified.

3.1.6 Time Sensitive Network

Time Sensitive Networking (TSN) is a set of protocols and technologies designed to make messaging more deterministic and reliable on standard Ethernet. It has been invented primarily for industrial networks dependent on performance guarantees on latency, minimization of jitter, and accurately controlling the timing of packets on the network. It includes elements like traffic prioritization, shaping, and Quality of Service mechanisms. An essential feature is time synchronization which enables devices to synchronize their clock to a precise reference. TSN networks are based on the IEEE 802.1 standard and aim to improve the network convergence between IT and OT systems.

3.1.7 SCADA systems

Supervisory Control and Data Acquisition (SCADA) systems are designed to improve operational efficiency, enable remote monitoring and control, and enhance safety of industrial processes by being a central hub for the overall plant's real-time state. They combine communication from Programmable Logic Controllers (PLC), Human Machine Interface (HMI), and Databases for long-term storage and trending into a large interconnected system. Overall, they benefit modern industrial automation systems by being a central node where the operator can monitor and manage complex processes in real-time and respond to events and alarms.

3.2 OPC UA

Open Platform Communication (OPC) was first released in 1996 with the intention of abstracting away the communication protocols of industrial controllers, enabling products from different vendors to exchange data. Initially, it was based on Microsoft DCOM technology, limiting its scope of use. In 2008, the protocol expanded to become OPC Unified Architecture (OPC UA), which is no longer restricted to only Microsoft systems and has interoperability and data modeling as its two fundamental pillars. OPC UA is not a single protocol but a stack with flexibility at every level, designed to be future-proof by exchanging individual protocols of the stack as new technologies develop. OPC UA has solid industrial support and is widely implemented as the solution for industrial controllers to enable cross-platform interoperability. It is designed for real-time, secure, robust, and platform-independent communication.

Chapter 3.2 is a condensed version of the OPC Foundation documentation [20].

3.2.1 Client – Server architecture

OPC UA is built on the client-server architecture, where the clients initiate requests for data to OPC server endpoints, which then respond by sending back the requested information or performing some action or method requested by the client. OPC servers are often programmable logic controllers (PLC) that directly control and interface the edge sensors and actuators. Clients can be Human Machine Interfaces (HMI) or analytics software requiring factory floor data. OPC UA communication is stateful, meaning a static and secure communication channel must be established and maintained between client-server pairs. The server URL identifies several protocol choices for each function of the channel.

Transportation is by UA TCP, an extension of regular TCP / IP protocol that adds a small overhead to optimize size, security, and maintains sessions during connection interruptions. The other popular choice is HTTPS, which establishes a secure Transport Layer Security (TLS) connection between devices.

Security is handled by UA-SecureConversation protocol, establishing a secure channel inside the transportation channel to maintain confidentiality and integrity. Implemented as a separate protocol is part of future-proofing OPC UA by enabling exchanging only parts of the protocol stack. The security level can be different on different endpoints and is included in the endpoint description. The main session security parameters are Security Policy and Security Mode.

Security Policy
- None – Sent as clear text
- RSA – Asymmetric encryption
- AES-256 – Symmetric encryption

Security Mode
- None
- SignOnly
- SignAndEncrypt

Figure 3 OPC security settings

Encoding messages is the serialization of complex data structures for communication on the wire. The most utilized protocol is UA BINARY, optimized for size and transportation efficiency. Other choices are JSON or XML, widely used because of their interpretability and being understandable to humans, but with less efficiency.

UA BINARY

```
1001101001010101001010110010111
```

JSON

```
"NodeId": {
   "IdType": 1,
   "Id": "MotorATemperature",
   "Namespace": 2
   }
```

XML

```
<opc:Namespace Prefix="Opc.Ua">
   http://opcfundation.org/UA/
</opc:Namespace >
```

Figure 5 OPC encoding formats

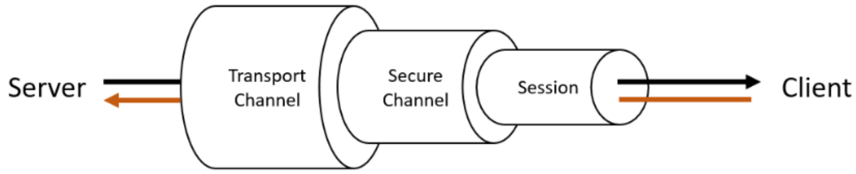


Figure 4 Protocol tunneling of lower level protocols.

3.2.2 Service-oriented architecture

The device interaction is based on a request-response and service-oriented architecture (SOA). Services are used to establish communications and to perform operations on nodes to enable information exchange. Every service request must be made within an established channel. Lower layers of the protocol stack ensure that data encoding, encryption, and transportation are standardized before it reaches the service level, abstracting away the services' implementation details. There is no need to understand internal components implementation details, allowing developers to freely choose a programming language and internal server and client details. The OPC UA specification defines a set of 37 services that manufacturers can decide to implement. Services are categorized into service sets to better understand their purpose and are listed for reference but will not be explained in detail.

Table 1 Services defined in the OPC UA standard

Discovery service set	**Query service set**
• FindServers • GetEndpoint • RegisterServer	• QueryFirst • QueryNext
Session service set	**Attribute service set**
• CreateSession • ActivateSession • CloseSession • Cancel	• Read • HistoryRead • Write • HistoryUpdate
SecureChannel service set	**Method service set**
• OpenSecureChannel • CloseSecureChannel	• Call
NodeManagement service set	**MonitoredItem service set**
• AddNodes • AddReferences • DeleteNodes • DeleteReferences	• CreateMonitoredItems • ModifyMonitoredItems • SetMonitoringMode • SetTriggering • DeleteMonitoredItems
View service set	**Subscription service set**
• Browse • BrowseNext • TranslateBrowsePathToNodeIds • RegisterNodes • UnregisterNodes	• CreateSubscription • ModifySubscription • SetPublishMode • Publish • Republish • TransferSubscription • DeleteSubscription

Error handling is an essential aspect of industrial communications. A set of standardized error codes constituting 16 bits have been pre-defined by the OPC Foundation and are included in the service response, simplifying error handling between devices from different vendors.

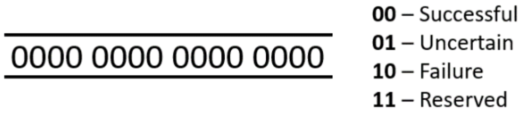


Figure 6 OPC Status code. First two bits are standardized.

3.2.3 Address space

OPC UA servers are divided into namespaces, each with an endpoint that specifies transportation protocol, encoding, and security. The NodeId and BrowseName need to be unique within each Namespace. Namespaces are stored in an array on the server index by number to minimize the size of the node identifiers. The OPC Foundation defines the first two indexes. “http://opcfundation.org/UA” is defined as index 0 and contains the base nodes defined by OPC UA. Server URI is index 1 containing server-specific nodes such as diagnostics information and certificates. Other namespaces are server specific and contain node types and object instances. Some industries are common to many and have been standardized in companion specifications. These specifications offer agreed-upon namespaces, including information models for common objects, such as the robotics industry. Examples of namespaces can be:

ns=http://opcfoundation.org/UA/Robotics/

ns=http://www.siemens.com/siemens-s7-opcua

3.2.4 Nodes

The fundamental building block in the server address space is a node. Nodes are object-oriented entities consisting of attributes with corresponding values. All nodes have a node class that specifies their attributes, and the base node class defines a minimum number of attributes mandatory to all nodes. Complex data models can be created by combining base node classes and references which help contextualize and link together their data. A client connected to a server can navigate the node structures and references to identify how the information models not previously known are built up. All nodes are uniquely identified by their NodeId, which comprises a node name and the Namespace it belongs to. An example of a node identifier in the namespace http://opcfundation.org/UA is:

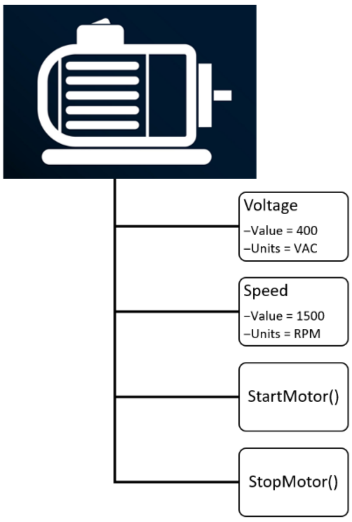
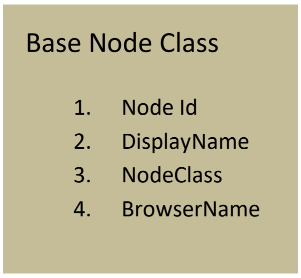


Figure 7 Example of motor object built as a node hierarchy

ns=http://opcfundation.org/UA;string=Temperature

The OPC Foundation defines eight basic node classes from which all other nodes are built. Type nodes represent metadata that explains and contextualizes the structure of the information. Instance nodes contain the variable's data and make up the majority of the address space.



Base Node Class

1. Node Id
2. DisplayName
3. NodeClass
4. BrowserName

Figure 8 OPC Base node class

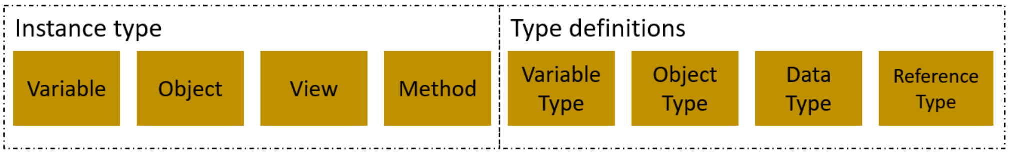
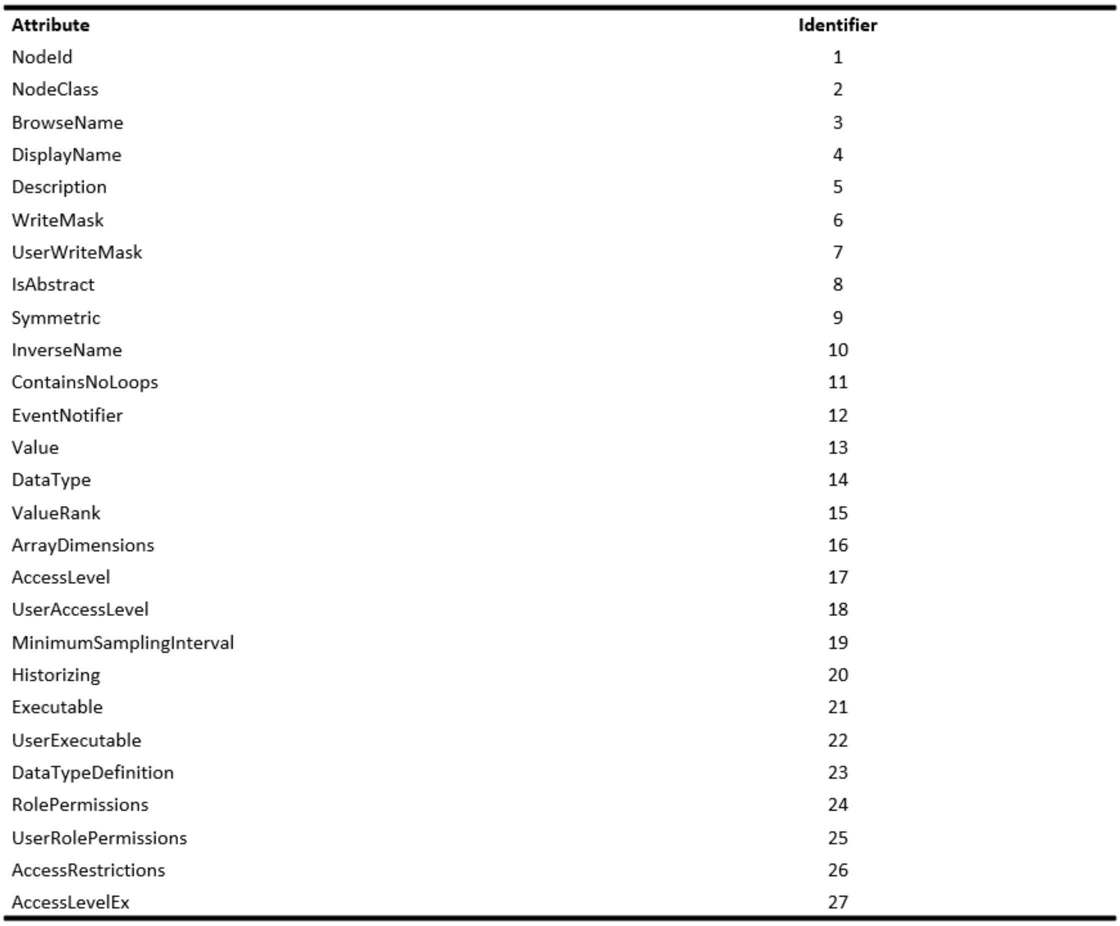


Figure 9 Basic node classes defined in the OPC UA specification

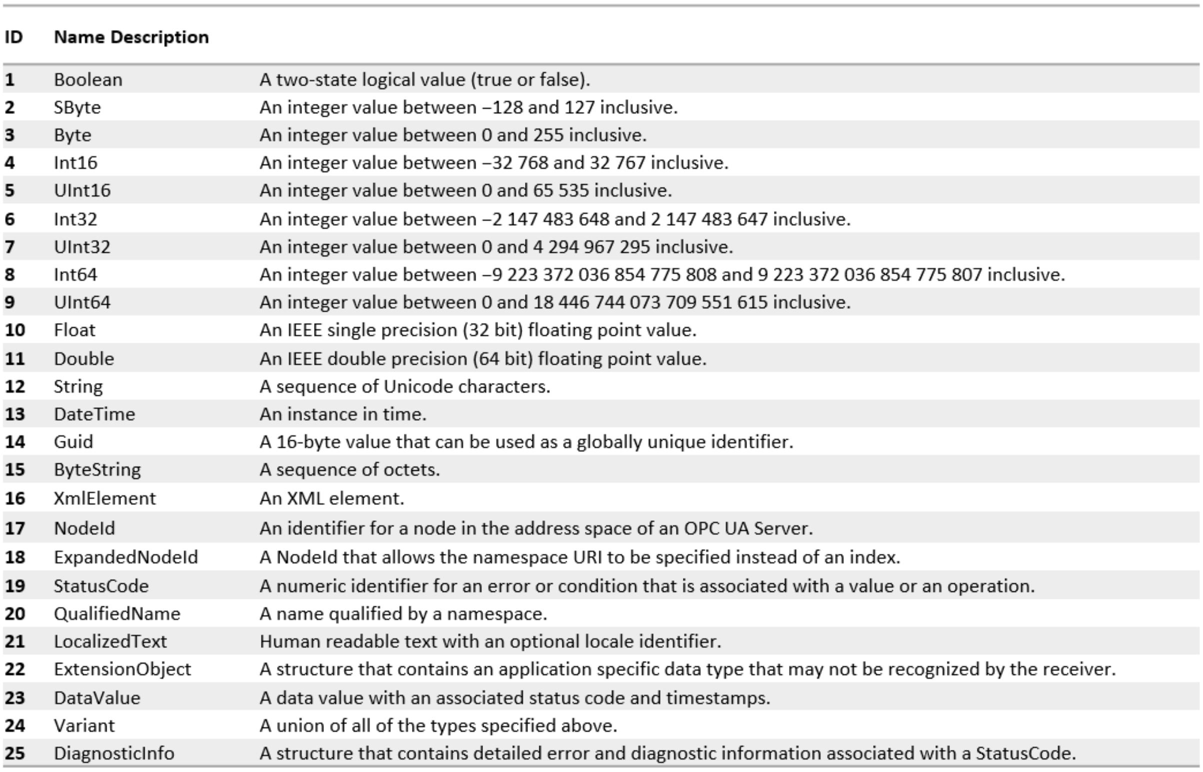
The nodes' data are stored in attributes, each with its data type. The OPC Foundation standardizes attributes and Built-in Data Types. User-defined datatypes are composed of the basic types and reference back to one of them.

Table 2 OPC UA defined identifiers assigned to attributes



Attribute	Identifier
NodeId	1
NodeClass	2
BrowseName	3
DisplayName	4
Description	5
WriteMask	6
UserWriteMask	7
IsAbstract	8
Symmetric	9
InverseName	10
ContainsNoLoops	11
EventNotifier	12
Value	13
DataType	14
ValueRank	15
ArrayDimensions	16
AccessLevel	17
UserAccessLevel	18
MinimumSamplingInterval	19
Historizing	20
Executable	21
UserExecutable	22
DataTypeDefinition	23
RolePermissions	24
UserRolePermissions	25
AccessRestrictions	26
AccessLevelEx	27

Table 3 OPC UA defined built-in data types



ID	Name Description	
1	Boolean	A two-state logical value (true or false).
2	SByte	An integer value between −128 and 127 inclusive.
3	Byte	An integer value between 0 and 255 inclusive.
4	Int16	An integer value between −32 768 and 32 767 inclusive.
5	UInt16	An integer value between 0 and 65 535 inclusive.
6	Int32	An integer value between −2 147 483 648 and 2 147 483 647 inclusive.
7	UInt32	An integer value between 0 and 4 294 967 295 inclusive.
8	Int64	An integer value between −9 223 372 036 854 775 808 and 9 223 372 036 854 775 807 inclusive.
9	UInt64	An integer value between 0 and 18 446 744 073 709 551 615 inclusive.
10	Float	An IEEE single precision (32 bit) floating point value.
11	Double	An IEEE double precision (64 bit) floating point value.
12	String	A sequence of Unicode characters.
13	DateTime	An instance in time.
14	Guid	A 16-byte value that can be used as a globally unique identifier.
15	ByteString	A sequence of octets.
16	XmlElement	An XML element.
17	NodeId	An identifier for a node in the address space of an OPC UA Server.
18	ExpandedNodeId	A NodeId that allows the namespace URI to be specified instead of an index.
19	StatusCode	A numeric identifier for an error or condition that is associated with a value or an operation.
20	QualifiedName	A name qualified by a namespace.
21	LocalizedText	Human readable text with an optional locale identifier.
22	ExtensionObject	A structure that contains an application specific data type that may not be recognized by the receiver.
23	DataValue	A data value with an associated status code and timestamps.
24	Variant	A union of all of the types specified above.
25	DiagnosticInfo	A structure that contains detailed error and diagnostic information associated with a StatusCode.

3.2.5 Publish and Subscribe

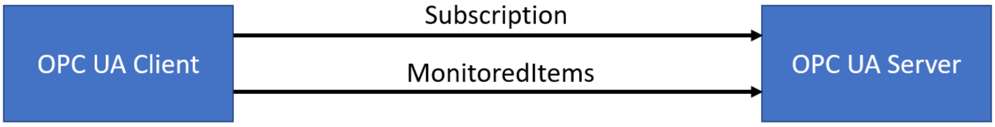


Figure 10 OPC UA subscription principle

In contrast to polling a server with request-response communication, a more elegant and recommended method is creating a subscription. When a client initiates a subscription, selected variables are specified in a "Monitored Items" list, and then the server publishes these variables when they change. Subscriptions significantly reduce the amount of network traffic because clients no longer have to poll servers with short intervals to be sure that updated values are transferred frequently enough. It must be noted that subscriptions are between individual client-server pairs. Separate subscriptions must be created between every client and server that will exchange data.

3.2.6 PubSub

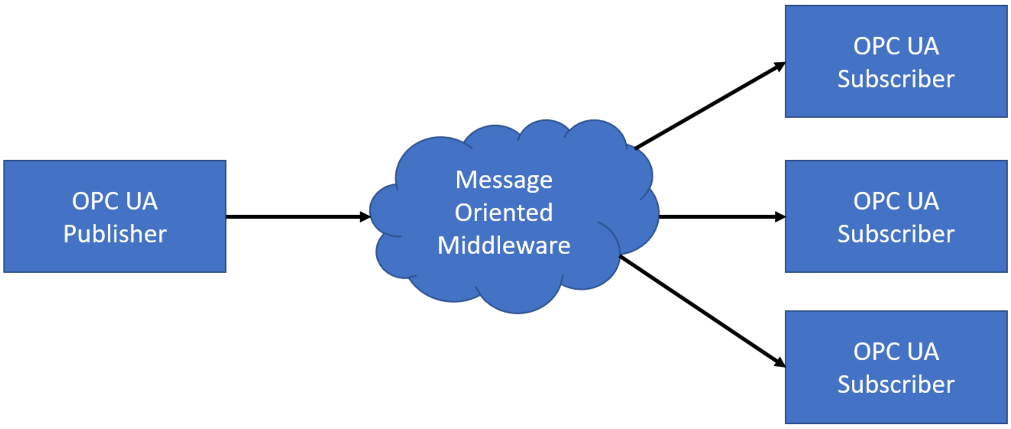


Figure 11 OPC UA PubSub principle

Client-server communication does not scale because of all the active connections that must be maintained between communicating devices. Therefore, the OPC Foundations have introduced PubSub, a One-to-Many and Many-to-One communication protocol, no longer a service-oriented protocol between clients and servers. A publisher, a source of data, will be configured to publish variables on change or on a fixed interval to subscribers, which consumes data through a message-oriented middleware. Two types of middleware are defined, Broker-less and Broker-based. If the transport protocol is UDP, then the message-oriented middleware only consists of routers and switches between the publishing and subscribing nodes. Variables will be broadcasted once with no guarantees that the recipient received them. Another option is to use MQTT protocol and a message broker as middleware. The broker can be configured to a Quality of Service (QoS) level that guarantees that recipients will receive the messages, even if it was offline at the time of transmittal. The main advantage is the decoupling of producers and consumers, allowing the network to be connected to the cloud and scale in order to be a fully interconnected IIoT infrastructure.

3.3 MQTT

Message Queuing Telemetry Transport (MQTT) is a lightweight messaging protocol that adds minimal overhead to guarantee message delivery and device status maintenance used for information exchange between an information publisher and subscriber. The primary goal of MQTT is to enable information exchange over limited bandwidth, unreliable, and high-latency networks, using one-to-many and machine-to-machine communication. The MQTT protocol was invented in 1999 by Andy Stanford Clark and Arlen Nipper. They required a protocol for minimal battery loss and minimal bandwidth to connect with oil pipelines via satellite. In October 2014, the MQTT protocol became an officially approved OASIS standard.

The MQTT protocol is built as a service-oriented architecture, making it possible to include clients developed in different programming languages into the same MQTT architecture. The protocol is designed to run over TCP/IP networks and comes with three Quality of Service (QoS) levels for messages. QoS is used to configure the reliability of the message delivery on the network.

Chapter 3.3 is a condensed version of the OASIS MQTT documentation [21].

3.3.1 Data broker

An MQTT data broker is at the heart of the MQTT protocol. The whole network depends on having an MQTT broker to route the traffic in the network. Having the broker as the central hub enables all clients in the network to be independent of each other, making the whole system highly modularizable.

An MQTT broker is a server that acts as a message hub for connected MQTT clients. The broker receives and manages the flow of messages to and from clients, temporarily storing them until they are delivered to the appropriate recipients.

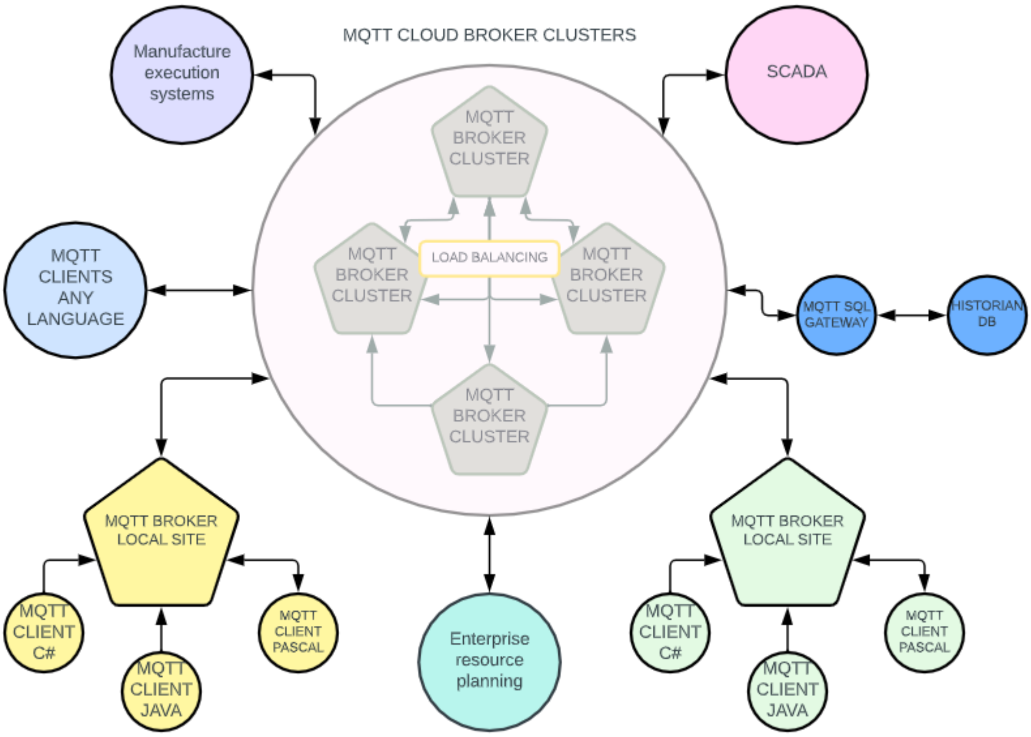


Figure 12 Clients and Brokers arranged in a MQTT architecture.

There is a risk if the broker that routes all client messages should malfunction. However, in MQTT networks, broker clustering and load-balancing techniques make it possible to ensure data flow even if a broker goes offline. Clustering also makes the MQTT broker network extremely scalable. If more computing resources are needed, you can add brokers to the cluster to expand the system.

3.3.1.1 MQTT 5.0

MQTT protocol has evolved, where 5.0 is the latest standard. MQTT 5 is not backward compatible with MQTT 3.1 and MQTT 3.1.1, however MQTT 3.1.1 is a minor revision being backward compatible with 3.1. Nevertheless, broker solutions are available such that all three standards can intercommunicate on the same network.

The major functional incentive for the standard revision to 5.0 was

- Improvements in scalability and large systems
- Improved error reporting
- Formalize common patterns, including discovery and request-response
- User properties
- Performance improvements

Changes were achieved by adding 42 properties that can be assigned to various control packages to give additional context and 43 reason codes to indicate the result of operations, such as server status messages and the cause of disconnection, which has greatly improved debugging.

3.3.2 Topics and Payloads

The "topic" is fundamental to MQTT. It is a string that represents the subject of a message. Devices publish messages on topics to which other devices subscribe to receive the information. In the MQTT network, these messages are referred to as payloads, much like PubSub for OPC UA. This allows for a flexible, decoupled communication model where devices can publish and subscribe to the information as needed.

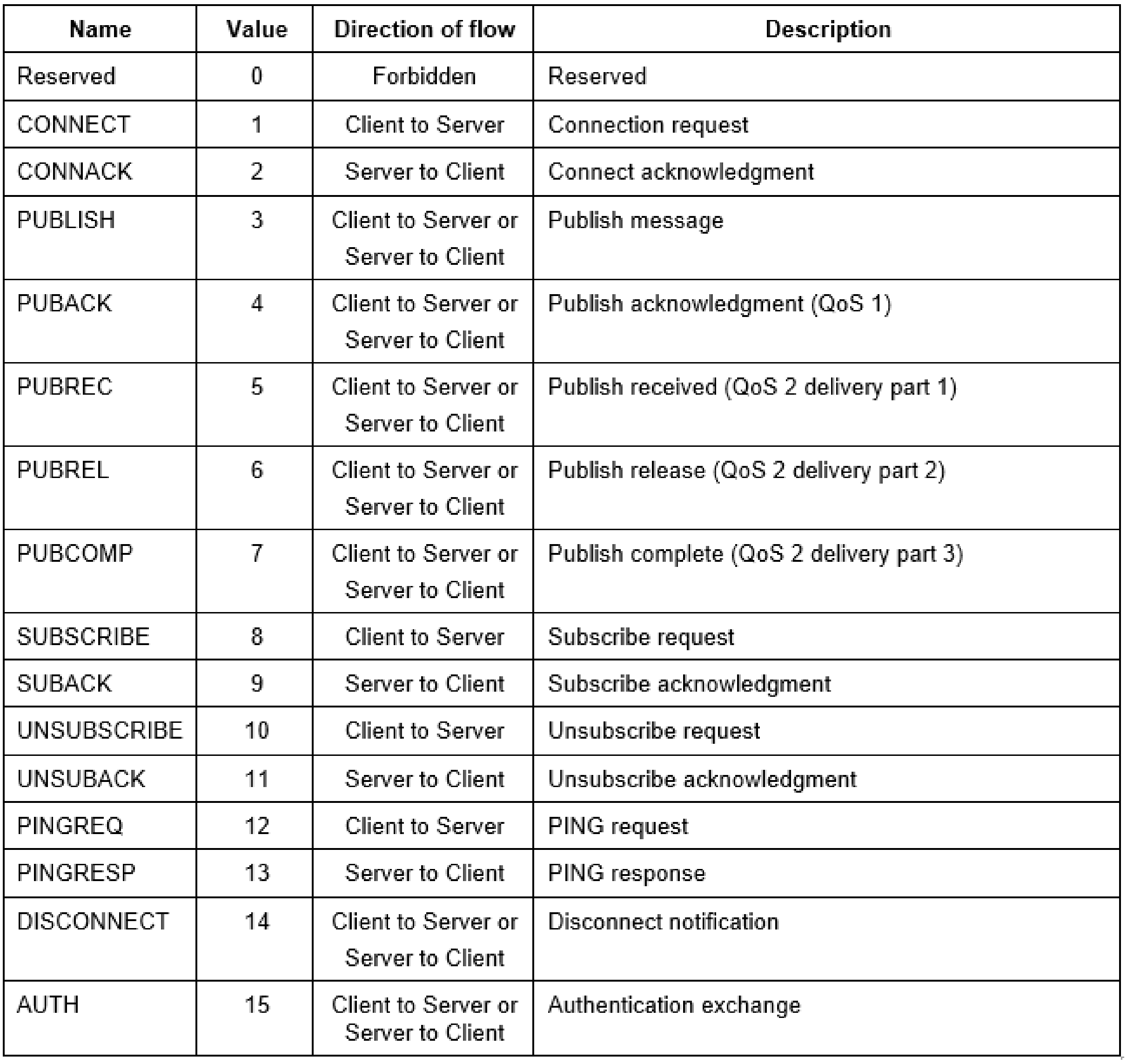
One disadvantage of MQTT is that there is no enforcement on how the message payload should be sent or received. Consequently, plug-and-play interoperability between devices can be challenging. A specification must be added to the payload to ensure plug-and-play. The Eclipse Foundation has attempted to resolve this shortcoming by developing the SparkplugB specification, described in chapter 3.4.

3.3.2.1 MQTT defined packages

The MQTT standard has gained popularity as a communication protocol for IoT devices due to its lightweight structure. The MQTT messages consist of three layers. The first layer of the MQTT control packets is a fixed header that contains essential information, including the message type, quality of service (QoS), and control flags. The second layer contains the topic name, packet identifier, and a custom field, while the third layer is the payload. The first layer is mandatory for all MQTT messages, ensuring the message is delivered correctly with the desired QoS.

The MQTT standard includes 15 different control packets that manage communication in the network, and they are available in both MQTT standards 3.1.1 and 5.0. MQTT has size limitations in place for topics and messages to ensure the efficiency of the protocol. A topic string can be up to 65536 bytes in length, while the message size cannot exceed 268435455 bytes, which is approximately 260MB.

Table 4 Types of MQTT control packages



Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Connection request
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment (QoS 1)
PUBREC	5	Client to Server or Server to Client	Publish received (QoS 2 delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (QoS 2 delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (QoS 2 delivery part 3)
SUBSCRIBE	8	Client to Server	Subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server or Server to Client	Disconnect notification
AUTH	15	Client to Server or Server to Client	Authentication exchange

3.3.3 Quality of service(QoS)

MQTT is implemented with three qualities of service levels for message delivery. QoS ensures stable message deliveries in unreliable networks. It also set a precedence on how often the same message should be retransmitted to prevent old information from arriving multiple times to the same client.

QoS0 "At most one". Message loss can occur, and delivery is after the best efforts of the operating environment. The messages are not stored or re-transmitted in the event of devices being offline and therefore provides the same guarantee as the underlying TCP protocol.

QoS1 "At least once". Messages are assured to arrive, but duplicates can occur. The sender stores the message until it receives a PUBACK packet from the broker, acknowledging the receipt of the message.

QoS2 "Exactly once". Messages are assured to arrive exactly once. Requires a four-part handshake between client and broker, where the packet identifier of the original PUBLISH message is used to coordinate the delivery of the message.

3.3.4 State awareness

The CONNECT package has a Keep alive parameter to maintain a state awareness of the network, determining the maximum allowed time between a Client last transmitting a message and when the broker determines that the client is no longer connected. If the time exceeds one and a half times the Keepalive period, the broker will publish the last will messages of the device. The minimum package that must be sent is a PINGREQ packet to ensure the keep alive tag on the connection. Edge devices can also be programmed such that the connection is deemed broken if no PINGRESP is received within a reasonable time after transmitting a PINGREQ.

3.3.5 Will flag

Devices can provide a will message when connecting to a broker. The will message contains a topic and a payload provided in the CONNECT package. It is stored on the broker associated with the client's sessions. The will message is published to subscribers of the will topic if the connection to the edge device is lost due to:

- An I/O error or network failure detected by the Server.
- The Client fails to communicate within the Keep Alive time.
- The Client closes the Network Connection without first sending a DISCONNECT packet with a Reason Code 0x00 (Normal disconnection).
- The Server closes the Network Connection without first receiving a DISCONNECT packet with a Reason Code 0x00 (Normal disconnection).

3.3.6 Clean sessions

MQTT was developed for an unstable network environment where connections were expected to fail occasionally. The clean session flag was implemented to enable a client to resume the previous session after a period of being disconnected, including all the session parameters.

3.4 Sparkplug B

Sparkplug B is an open-source software specification that defines a set of rules and guidelines for how data should be formatted and organized when it is sent over MQTT, including the specification of topic structure and payload formats. The specification provides MQTT clients the framework to enable plug&play interoperability in a range of devices that is optimized for the SCADA/IIoT solutions. Chapter 3.4 highlights the significant aspects given in the specification [6]

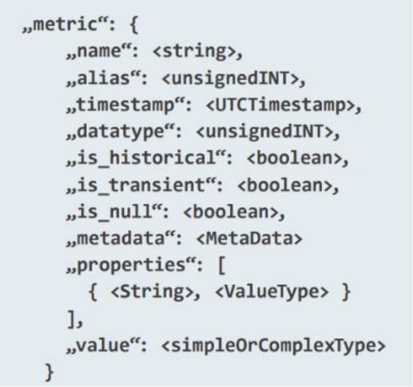
The three main goals that SprakplugB intends to resolve are

- Define an MQTT topic Namespace
- Define MQTT state management
- Define the MQTT payload

Sparkplug requires a 100% implementation of minimum MQTT standard 3.1.1 on clients and brokers. An advantage is that it is possible to send regular MQTT traffic on the same network as SprakplugB. The only implication is that clients not supporting SparkplugB cannot interpret those messages.

3.4.1 Benefits of Adoption

One of the biggest advantages of implementing SparkplugB is that it provides an agreed-upon data structure to transmit and receive information between devices. The specification defines a .proto file, a blueprint of an information model with standard attributes and defined data types, which also allows for nested data structures providing the necessary flexibility to send complex models. By making the essential attributes mandatory and including additional attributes as optional, SparkplugB has achieved a balance between generality and optimized processing by the receiver while restricting the freedom needed to achieve reduced integration time and costs.



```
„metric“: {
    „name“: <string>,
    „alias“: <unsignedINT>,
    „timestamp“: <UTCTimestamp>,
    „datatype“: <unsignedINT>,
    „is_historical“: <boolean>,
    „is_transient“: <boolean>,
    „is_null“: <boolean>,
    „metadata“: <MetaData>
    „properties“: [
      { <String>, <ValueType> }
    ],
    „value“: <simpleOrComplexType>
  }
```

Figure 13 Example of SparkplugB attributes [6].

State management is another essential feature that makes MQTT more geared toward the requirements of a SCADA system. By building on existing MQTT functions such as will message and standardization of topic namespace to define topic areas for NBIRTH and NDATA messages, SparkplugB has standardized data transfer, device discovery, and online state management.

Another benefit is the report-by-exception principle which Sparkplug is based on. In traditional poll-response systems, a data consumer must poll the data producer for information to check if data has changed. In report-by-exception, the data producer publishes only when the data changes to save bandwidth, computing power, and memory consumption. State management and death certificate are used to detect stall data and immediately notify the data consumer of any abnormal behavior, like a disconnect.

3.4.2 Death and birth certificates

SparkplugB utilizes the built-in functions of MQTT will message and keep alive and adds a set of defined BIRTH and DEATH topic namespaces and payload definitions.

A birth and death certificate is sent immediately after a device has connected to the broker. The birth certificate is used for the management and discovery of the device. The content is up to the protocol user, but common practice is to announce all the information, data types, and last known values that the device will publish. It is possible to re-publish a new birth certificate if the information that the device sends should change while online.

Suppose a publishing device were to go offline after it has sent a birth certificate. The broker will then send all subscribing entities of the publishing device the death certificate of that device. The Death certificate is registered as the MQTT will message.

3.4.3 SparkplugB compatible systems

As of 19.01.2023, there are listed zero hardware systems and two software systems supporting SparkplugB on the Eclipse official site for Sparkplug. The software systems supporting SparkplugB are currently Eclipse Tahu and HiveMQ. However, several suppliers like EMQX MQTT system supplier and OPC UA system supplier claim SparkplugB compatibility, so it might be wider adopted than the official listing indicates.

3.5 Data structuring

3.5.1 ISA 95 standard

ANSI/ISA-95 standard [22] is a set of guidelines for implementing an enterprise-wide system with a well-thought-out tag structure that will prevent name collisions and which can be easily extended. The tag structure provides data context by labeling information according to where it was produced. The standard is intended to be extensive, describing the entire information flow from the factory floor all the way up to the business and management level. It is the only recognized standard for hierarchically naming the different parts of a business or manufacturing process. The standard also defines models, including standardized attributes for when information is transferred from the operational systems to the management layers.

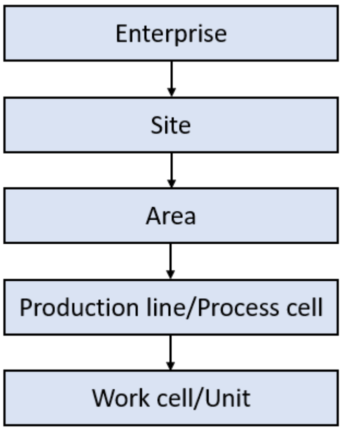


Figure 14 ISA95 Enterprise hierarchy

3.5.2 Standardization

Agreeing on common attributes, data types, and encoding optimizes data sharing between systems. It also reduces the potential for misinterpretations and subsequent errors. For example, Unix has a standardized time format and time zone used in timestamps, efficiently eliminating uncertainty at the receiver.

3.5.3 Tagging conventions

All nodes need to be assigned symbolic names to be uniquely identified. Two types of grouping variables are commonly used. Grouping by type is when similar information is arranged together. For example, all temperature measurements are given a tag number and grouped accordingly in a system. It is essential to have a sound tagging convention that prevents naming collisions and eventually results in the need to re-think and re-tag parts of the enterprise.

On the other hand, grouping can be based on the physical layout of the organization. Each temperature sensor belongs to a module that also has its own tag number. The complete temperature tag will be the entire link of the tag names from the enterprise root to the individual variable value. Physical tag numbering can be more forgiving because the same tag numbers can be re-used without concerns for naming clashes. A disadvantage is the length of the tag numbers. It is primarily a concern for those who design the tag system and do initial tagging. During communication, most protocols have functionality for exchanging unique variable handles between hosts to shorten the variable address.

3.6 Cyber security

National Institute of Standards and Technology (NIST) defines Computer security as *"The protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability, and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications)." [23] It is the tools and principles implemented to safeguard the information from being eavesdropped on, tampered or deliberately falsified.*

3.6.1 Confidentiality, Integrity, and Availability

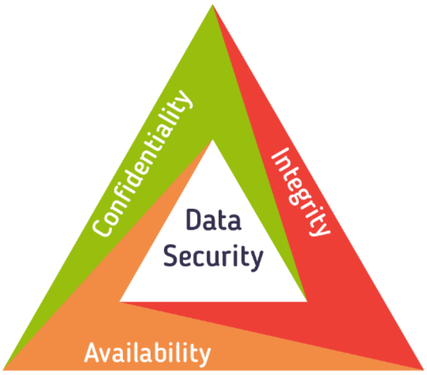


Figure 15 The CIA triad [5]

The security triad is a model used to describe the three fundamental principles within cyber security to ensure that information is protected and systems are available and reliable.

Confidentiality is the protection of information from being accessed or disclosed to individuals or systems not authorized for it. Security is achieved through the use of encryption, user authentication, and authorization. Confidentiality measures are mostly passive technologies like encryption that scrambles the data making it unreadable or restricting access to the communication channel.

Integrity is guarding against information tampering, modification, and the consistency of information. It builds on the principle that it should not be possible for anyone but the owner of a publicly known secret key to produce the digital signature accompanying the message. Measures to prevent or detect message modification are hashing, certificates, and digital signatures.

Availability refers to ensuring timely and reliable access to the systems and information therein. It is achieved through measures such as redundant systems, backups, load balancing, and recovery procedures. Availability measures must handle single point of failure and cyber attacks that aims to overwhelm the system making it unavailable to the intended users. A distinct difference between IT and OT systems is that availability is by far the most important factor for OT systems, which must be available at all times.

3.6.2 Authentication and Authorization

In short, authentication is the process of verifying a user or system, while authorization is the process of restricting information access to only individuals or systems that have been granted permission.

Authorization often requires the individual or system to provide a set of credentials that are verified against a database to confirm the true identity of the requestor. Examples of credentials used for authentication are username and password, certificates, or pre-shared keys (PSK), all of which should be kept secret. Multifactor authentication can be utilized when increased security is required by requiring several identifying factors like a username/password, something the individual knows, combined with a token like an authenticated cellphone that is something that the individual possesses.

Authorization are the measures that restrict access to resources or parts of the system. The principle of the least amount of access is when everything is initially blocked, and then access is granted based on necessity. Logging when and what information a user accessed or changed is also part of the authorization system. Often users have to authenticate before the system determines what information is made available based on registered access level. Maintaining and updating the individual's access permissions can be work intensive. Therefore a role-based access model assigning pre-defined roles with specific access levels configured is often chosen.

3.6.3 Cryptographic algorithms

Cryptographic algorithms are tools used to encode information making it unreadable to unauthorized individuals. A fundamental principle is that security should not depend on the secrecy of the implementation details because the revealing of these details must be anticipated. The security lies in the length of the encryption key and the algorithm's ability to randomize the data. It is essential that the encryption is reversible so that the recipient can decrypt the message. There are essentially two different types of encryption.

Symmetric encryptions utilize the same key for encryption and decryption. Symmetric algorithms mostly use very efficient permutations and substitutions, which also can be implemented in hardware for even faster processing times, making it suitable for large amounts of data. The most widely used symmetric algorithm is Advanced Encryption Standard (AES). The main disadvantage of symmetric encryption is the need to distribute secret keys between communicating participants.

Asymmetric encryption like RSA uses mathematical operations to transform the information. Two parties can create a common shared secret by sending each other partial mathematical products, enabling parties without prior knowledge to establish secure communication. Each participant will end up with a private and a public key pair. Data encrypted with the public key can only be decrypted with the associated private key. The algorithms are often computationally heavy, and the length of the secret key is significant to prevent brute force attacks making these algorithms unsuitable for large amounts of data. Asymmetric algorithms are primarily used for exchanging symmetric keys, authentication, and digital signatures.

3.6.4 Certificates

The secrecy of the cryptographic keys is essential to secure communication. Man-in-the-middle attacks are when an advisory intercepts communication between two parties during session establishment, impersonating the intended recipients and exchanging secret keys with both parties. All messages are then relayed through the advisory enabling him to eavesdrop on or modify the content of the communication. Digital certificates are asymmetric public keys registered and stored by certificate authorities (CA). The CAs have publicly known certificates used to sign the hashes of public keys together with a claimant's identifying information. The CAs must adequately verify the claimant before the certificates are distributed. If the signing CA is trusted, the entire chain of certificates and signings can be verified before secret keys are exchanged by encryption with the recipient public key. X509 is a certificate standard and is the most widely used format for digital certificates. Most operating systems (OS) and internet browsers are distributed with trusted root certificates pre-registered.

3.6.5 Certificates stores

A certificate store is a secure database of digital certificates and possibly the associated private keys. It’s commonly subdivided into a personal store containing certificates used by the host, a trusted store for known and verified partner devices, and an untrusted store containing certificates of known malicious devices. An application can have its own certificate store, or alternatively, it can be shared, such as the Windows certificate store to which many applications interface. The big advantage is that collecting all certificates in one location provides better organization and improved security since mechanisms such as secure storage and preventing memory leaks are implemented around this controlled storage location. A certificate store contains numerous certificates, possibly from different certificate authorities. Certificates signed by a root CA will be trusted as long as the root CA is stored in the certificate store.

3.6.6 Hashing

A hashing algorithm is used to transform a variable-length message to a fixed-length fingerprint of the original message. A good hash function should map the input space to the output space evenly and with the same probability. Cryptographic hash functions are pre-image resistant requiring that it be infeasible to find the input x given the hash h(x) and strong second pre-image resistance requiring it to be infeasible to find any two inputs x and y that map to the same output h. Digital signatures often utilize a hash function combined with a secret key to create a message identifier attached to the message. Message integrity is achieved because only the owner of the secret key could have produced the hash. Alternatively, a hash can be encrypted with a private key as a digital signature. It can only be decrypted with the corresponding public key, proving that only the one possessing the private key could have produced the message. In principle, encrypting the entire message using the private key will prove integrity but is computationally slow. Only encrypting the hash saves a lot of time.

3.6.7 Transport Layer Security

The most used encryption standard on the web is Transport Layer Security (TLS) secures data to be transmitted at the transport layer of the OSI model by creating a secure channel from port to port. TLS encrypts data from the application layers so that applications do not need to implement security protocols to communicate safely on the network, making it a popular choice for web browsers and servers. An example is Hyper Text Transfer Protocol Secure (HTTPS), which is regular text encapsulated by TLS and recognized by the famous hallmark lock symbol in the address bar of internet browsers. Secure Socket Layer (SSL) is the predecessor of TLS and is still widely used even though TLS was intended to replace it. These protocols are essentially the same. Authentication of the recipient and exchange of cryptographic keys are handled with very little overhead by the 3-way handshake and SSL/TLS certificates. TLS traffic uses the standard port 443, which usually is allowed to pass most firewalls because it is authenticated and encrypted communication.

3.6.8 SSH

Secure Shell (SSH) is a network protocol that allows two computers to connect with each other and remotely access the terminal of another computer. This connection provides a secure channel over an unsecured network by encrypting all communication. SSH also provides the ability to view the file structure on the system. During this thesis, we utilized an extension in the Visual Studio Code IDE called Remote Explorer. This extension provides a shell terminal, file explorer, and IDE editor for instantaneous code editing.

3.6.9 VPN

Encryption and data integrity between two endpoints can be implemented at the IP level of a network as Virtual Private Network (VPN). Data is secured in a secured tunnel established with encryption from router-to-router or host-to-router, providing a path secured against manipulation in an unsecured network. VPNs implemented in router endpoints can be very efficient since traffic on the internal network is not encrypted by the hosts, but traffic destined for the remote endpoint will be encrypted by special-purpose hardware optimized for efficiency and security. [24]

3.6.10 IPSec

IPsec stands for Internet Protocol Security and is a VPN protocol used to provide secure communication over an insecure network, such as the internet. IPsec can be used to create VPNs between two endpoints, allowing them to communicate securely as if they were on a private network. IPsec can be used in two modes: transport mode and tunnel mode. In transport mode, only the payload of the IP packet is protected, while in tunnel mode, the entire IP packet is protected. Tunnel mode is typically used when creating VPNs between networks, while transport mode is used for end-to-end communication between hosts.

3.6.11 Firewalls

Firewalls protect the network by filtering incoming and possibly outgoing traffic across the network's border. Inbound and outbound rules are set up to decide on what criteria traffic is blocked or allowed. Stateful firewalls increase security by only allowing devices inside the protected network to initiate communication across the border. A record with a configured timeout keeps track of ongoing sessions, only allowing outside traffic to enter if it belongs to an active session. Firewalls are often implemented in-depth, segmenting the network into different security zones with different rules for traffic entering different segments. Servers that must be available for hosts on the outside network are often located in the outermost segment, called a demilitarized zone. Traffic destined for these servers does not need to enter the inner network, reducing the risk of malware spreading to these systems. If the devices inside the network are only transmitting, not receiving any data from the outside, then all incoming traffic can be blocked, significantly increasing network security.

3.7 Web technology

“Web Technology refers to the various tools and techniques that are utilized in the process of communication between different types of devices over the internet." [25] Communication throughout the web is managed by web-browsers, which translate, create, deliver and manage web-content using a hypertext markup language (HTML). We can send a request to a web server with a browser that responds with HTTP.

3.7.1 HTTP

Hypertext Transfer Protocol (HTTP), is an application-layer(OSI) request-response protocol that transfers data over a network. HTTP was designed for communication between web browsers and web servers. Due to its extensibility, it can fetch anything from hypertext documents to images and videos. It can also be used to post content to servers or update web pages on demand. HTTP has changed and been updated slowly since its creation, which has been a coordinated effort launched by the Internet Engineering Task Force (IEFT) in 1996. In later years, many websites made the switch to HTTPS, which is HTTP with encryption and verification. HTTPS uses TLS to encrypt and sign normal HTTP requests and responses.

3.7.2 API

API stands for Application Programming Interface. Its purpose is to open up the services a program offers to others while still being secure and only granting requests it deems legal by pre-written rules. An API could allow us to interact with a service and gain some access to the service's data without being constrained by the possible limitations of the service's own interface.

3.7.3 JSON

JSON, also known as JavaScript Object Notation, is a structured text file for formatting and structuring data for storage. JSON was designed to be very basic and straightforward and to closely mimic data structures and primitive types that many programming languages have built-in by default. JSON excels at being easy to access and manipulate. Being able to freely exchange and work with stored data is nearly as important as the data itself.

3.7.4 XML

XML, also known as Extensible Markup Language, is a markup language and file format for storing, transmitting and reconstructing arbitrary data. It defines a set of rules for encoding documents in a format readable by humans and computers. While very much like JSON, it has a couple of distinct differences. The biggest one is that XML must be parsed with an XML parser, while a standard JavaScript function can parse JSON. XML must also include end-tags and a nested tag pair structure, making it longer and slower to read and write than JSON.

3.7.5 Simple Binary encoding

Simple Binary Encoding (SBE) is a binary-format protocol for decoding and encoding messages with low latency and deterministic performance. The SBE message format is specified using native primitive types, which means there is no need for further translation since these types are generic. It also only concerns itself with data representation, which means that the structure of the message is not subject to other applications. The message layout, which is specified in the SBE template, is based on XML.

3.7.6 Protobuf

Protocol Buffer or Protobuf [26], is a language-independent binary serialization format developed by Google. It is a flexible and efficient way to transmit structured data between different systems, particularly in high-performance scenarios. It is used by defining a schema in a ".proto" file. This file specifies the data structure, data types, and field names for all messages that will either be sent or received. Once a schema is agreed upon and defined, every receiver and transmitter receives the same proto file, which Protobuf uses to generate the needed structures in the code base to encode and decode messages. Protobuf messages are typically smaller and faster to transmit and parse than formats like XML and JSON.

3.8 Hosting Technologies

Hosting technologies refer to different methods of making applications and programs available over the Internet. In this thesis, we will be implementing cloud hosting technology that employs a cluster of servers, allowing for a high scalability level. Resources can be easily scaled up or down based on demand, ensuring efficient resource allocation.

3.8.1 Virtual Machines

Virtual machines (VMs) are an increasingly popular technology used in the IT industry for various purposes, including application deployment, testing and development. They allow multiple operating systems and applications to run on a single physical machine by creating a software environment that simulates the behavior of a physical computer. The VM consists of a software layer that uses the underlying physical hardware, including the CPU, memory, storage, and networking resources. This enables multiple VMs to run on a single physical machine, each with its own isolated environment and virtual resources, including the operating system, applications, and data. Another advantage of virtual machines is their ability to provide fast and consistent deployment of applications. By packaging the application, operating system, and other dependencies into a single VM image, developers can then easily deploy and test their applications on different environments without worrying about compatibility issues.

3.8.2 Clustering

Clustering involves grouping multiple servers to form a single, highly available system. The primary objective of clustering technology is to enhance the reliability and availability of services. Reliability is ensured by incorporating multiple servers that offer redundancy and failover protection. In case of a server failure, the workload is automatically transferred to another server. Moreover, availability is increased by leveraging load balancers to distribute incoming requests across the cluster of servers.

3.8.3 Loadbalancers

Load balancing is a technique used to distribute incoming requests to a group of servers to ensure that the workload is evenly distributed across the cluster. It works by placing the load-balancer as the entry point for clients approaching the cluster. When a client sends a request to an application on the cluster, the load balancer receives this request initially and decides which server in the cluster should handle this request based on factors such as server capacity, availability, and current workload. In this thesis, we will use HAproxy loadbalancers[27].

3.8.4 Kubernetes

Kubernetes [28] is an open-source platform used for container management in a cluster. With Kubernetes, containers can automatically scale based on demand, and additional containers can be started up when traffic increases and removed when traffic decreases. This ensures that computing power isn't wasted on maintaining unnecessary applications and that there is always enough capacity to handle an increase in traffic. One or more servers will act as the control panel which spawns and despawns containers. These servers are often referred to as masters. The masters assign containers to the worker nodes, which consist of other servers that have joined the Kubernetes cluster as worker nodes and are selected upon joining the cluster. Containers are a lightweight way to package applications and are defined by applying Kubernetes manifest files that describe the configuration of the container, networking, and its image. The container holds an image of the application it runs, created by building Docker files. When a single instance of these containers is deployed on the

Kubernetes cluster, they are referred to as pods. For this thesis, we will be using the K3S Kubernetes cluster software.

3.8.5 Docker

Docker is an open-source platform for containerizing applications [29]. It makes it possible to package applications and system dependencies into a container. Containers can then be built in whatever environment the developer chooses. The built containers are often referred to as images. Docker uses a Dockerfile to build the container, which is a script that defines the components of the image, including the application code, runtime libraries, and other dependencies the application code needs. Once the image is built, it can be pushed to the Docker registry as a private or public repository for easy access across platforms.

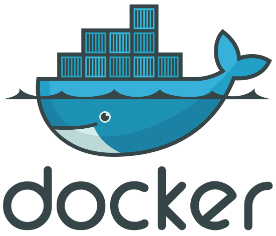


Figure 16 Docker logo [2]

3.9 ChatGPT

In November 2022, ChatGPT was introduced to the world. This large language model was created using machine learning techniques to mimic human-like responses in written form. In the initial phase of this bachelor thesis, we experimented with this new technology. While ChatGPT can produce perfectly formulated texts, we experienced that it often contains inaccuracies in the information it provides.

Figure 17 ChatGPT logo [3]

However, ChatGPT excels at taking loosely formulated written text, written as our thoughts appear, and restructuring it into a more formal sentence structure while maintaining the accuracy of the information. This has greatly increased our efficiency in writing texts, as we can now focus on conveying information rather than spending vast amounts of time scouring through our texts for sentence structure and grammar errors.

<u>An example of how ChatGPT has been used in this thesis, under is a loosely structures text:</u>
The need for VPN technology is paramount in order to ensure isolation of the local area network where the factory brokers are located. This structure of isolating the local sites is often used by companies today.
<u>What ChatGPT suggests from the loosely written text.</u>
The use of VPN technology is essential to ensure isolation of the local area network where the factory brokers are located. This approach of isolating local sites is commonly employed by companies today.

4 The realization of the Unified Namespace

Chapter 4 will explain how our team started with a simple broker concept and expanded it into a scalable networked broker solution.

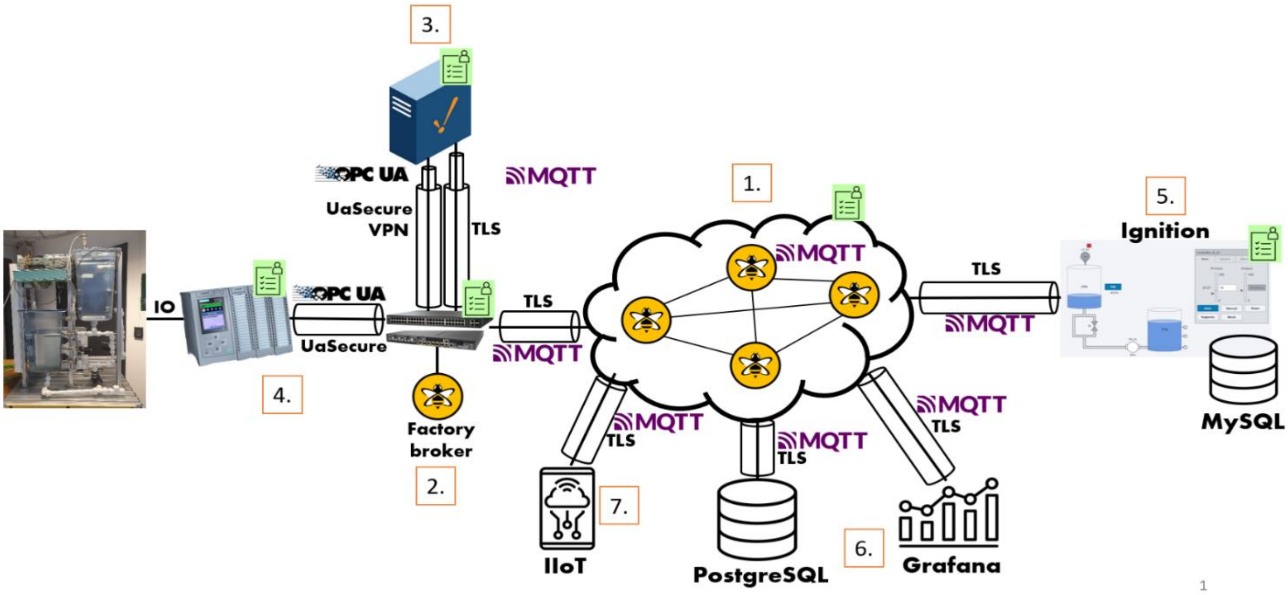


Figure 18 The final UNS network which was implemented in this thesis.

The network solution has a central component, which is the cluster broker (1.). This broker acts as the heart of the system, connecting all factory brokers into a large network. The factory broker (2.) is a standalone broker that operates at local sites such as factories. While they can function as independent local entities, they can also share their clients with the entire network by connecting to the cluster. The factory broker (2.) can have different edge devices such as PLCs (4.) that can share data with SCADA systems using protocol gateways like Ignition (3.) through VPN connections to the factory site. These devices are connected to the factory broker locally, but through the cluster, can be monitored remotely by offsite SCADA systems (5.). Data in this network is volatile. Therefore, storing the data for long-term access is crucial, which is the responsibility of the Historian (6.) The final component of the network is an IIoT(7.) device, that we have self-designed and produced to experiment with connecting non-standard hardware to architecture.

The network in the figure utilizes the MQTT protocol for all information exchanges between the different entities in the network. The connections' security is ensured by using Transport Layer Security (TLS) and user authentication in the form of usernames and passwords. In the figure, the systems secured with usernames and passwords are marked with a green card.

Two network solutions support the network. The factory site is set up using a Cisco LAN with Cisco routers and switches connected to the internet, while the cluster is connected to the factory brokers and data users via an internet connection. These two technologies form the "glue" that binds the different entities in the network together. Overall, the network architecture is designed to ensure seamless communication between different entities while maintaining security and reliability.

This is only a short summary to give the reader an overview of what is implemented in the UNS network. Further details of the individual systems are covered in this chapter.

4.1 MQTT broker implementation

Chapter 4.1 will detail how we implemented our chosen MQTT broker solutions and our reasoning for these selections. When it comes to selecting an MQTT broker, numerous options are available, such as Mosquito, HiveMQ, EMQ, and AWS IoT Core. Many companies offer hosting services that use these brokers, ranging from standalone MQTT brokers to large cluster array broker solutions. Most of these brokers come with complete preconfigured systems, which eliminate the need for users to write config files in XML, set up cryptographic infrastructure, manage networking, and host and configure servers.

During the preliminary research for this project, it became evident that we would need to host our own broker. The main reason behind this decision was that by building our broker solution, with all the added complexity, we would gain a better understanding of MQTT broker solutions and be more capable of designing our own network. Additionally, it was done to tailor the broker solution to our specific needs rather than a general implementation.

The MQTT network we have designed is not limited to local area networks and is intended to be deployed over the Internet. Since the brokers are connected through secure encrypted communication tunnels, the location of the different sites is less significant, with the only requirement being a stable internet connection. The local servers will collect all of their underlying topic publishers and subscribers, merging them into one packet. This packet will then be transmitted to the cluster servers for handling. If no access control is enabled in the network, all topics from every site will be available to all participants in the network.

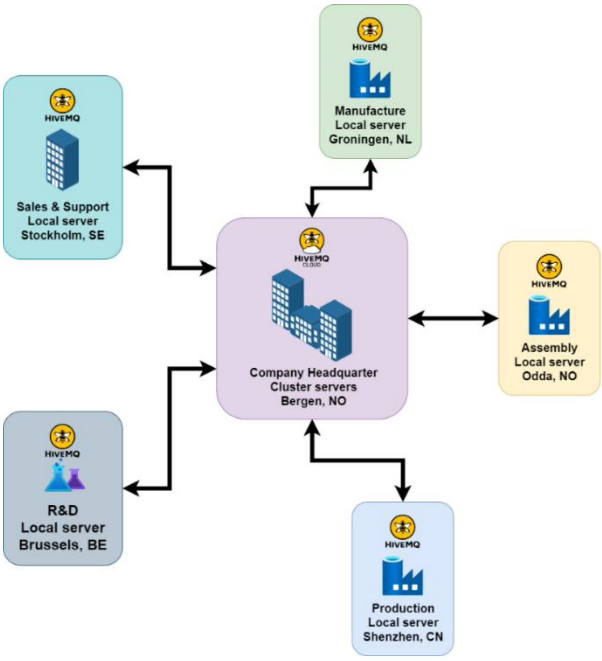


Figure 19 Global MQTT UNS network

During the preliminary project meetings with Goodtech, they wanted us to build and test the HiveMQ broker solution. As such, HiveMQ was the solution we ended up using in our Bachelor's thesis. We have built two broker solutions. The first solution is the standalone HiveMQ for local servers, which acts as an intermediary between all the edge devices on that particular site and the cluster.

The second solution is for the cluster server. This cluster contains multiple servers with brokers connected to enable scaling as needed. The cluster broker administers all clients connected to the UNS. The factory broker must make all its content available through the cluster network so that data subscribers across the network can access the factory topics.

4.1.1 MQTT Brokers

Both the factory site and the cluster broker use HiveMQ broker technology. This broker comes in two versions - the community version and the enterprise version. To connect the various brokers together, it was necessary to use the enterprise version of HiveMQ, which is not free for commercial use. However, a full enterprise broker can be used with a four-hour limitation for testing purposes. This provided our team the opportunity to test a full-scale HiveMQ implementation as a proof of concept without committing significant resources to funding. During this chapter, representations of technologies used are often displayed with logo of the brand, these logos are sourced from [2]. HiveMQ logos are sourced from [30].

The HiveMQ broker offers the ability to have an open MQTT connection and/or a Secure MQTT connection. To get Secure MQTT connections, TLS must be applied to the connection between the clients and the brokers. During our implementation, we initially used the open connection for testing purposes before applying secure connections to all MQTT clients. In order to make the factory site share its topics, we utilized a bridge extension for HiveMQ. In short, this bridge extension enables topic sharing between the cluster broker and factory broker such that the topics in the factory broker get transferred to the cluster and made available to the clients there.

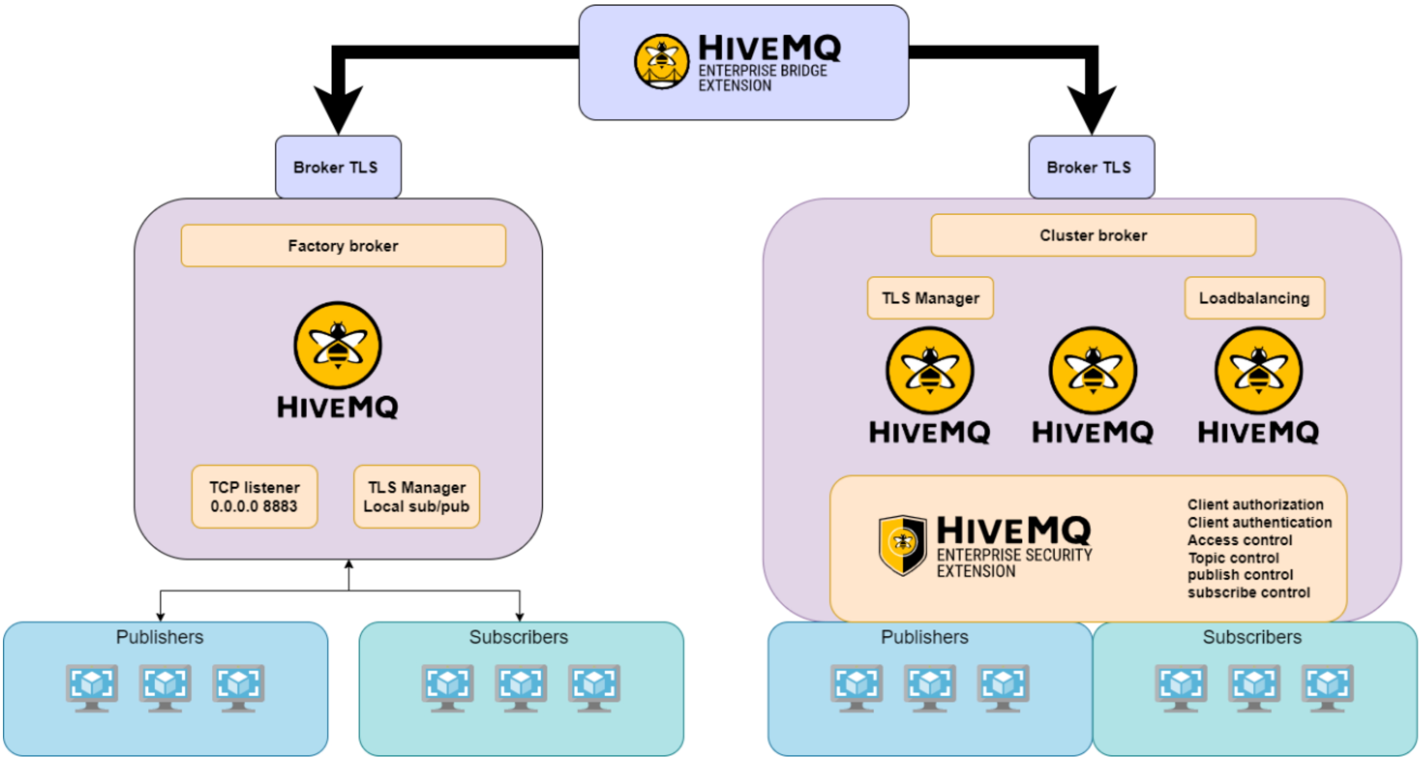


Figure 20 MQTT traffic bridge between local and cluster brokers

4.1.1.1 Operating Systems(OS)

Any server connected to any network must maintain its own operating system, which acts as an intermediary between the computer hardware and the software running on it. During the initial implementation phase, multiple operating systems were tested for viability. HiveMQ recommends a Red Hat enterprise Linux(RHEL) operating system. This OS was discarded due to high licensing fees but is a potential candidate for hosting UNS due to its high focus on security. Another operating system that showed promise was CENTOS. A broker solution was deployed on the CENTOS STREAM operating system in order to test the viability of the operating system. It became evident that CENTOS STREAM was a community project maintained and secured on a voluntary basis, becoming a concern that this could lead to a potential security risk in future deployment of UNS and was subsequently dropped. Ubuntu 2022 Server(https://ubuntu.com/) distribution is an OS with a dedicated team to address security issues and is continually updated. It is also a widely adopted OS, making it easier to implement software on it. That is why we choose the Ubuntu 2022 Server OS. For our cluster solution, deploying Ubuntu 2022 server OS on our Raspberry Pi hardware was not feasible since we would have exhausted a substantial amount of the available computing resources. This is only a limitation for our thesis project since we are building our cluster on Raspberry Pis. This would not be an issue if we had proper server hardware. We deployed Raspberry's own OS raspberry pi LITE 64-bit onto the Raspberry Pi's to sort out this limitation.

4.1.1.2 TLS

All our broker implementation employs a two-way authentication for TLS. When the TLS encryption is initiated, it's called a handshake. In short, the client sends a request to the server indicating that it wants to initiate a TLS connection. The server sends its certificate and a public encryption key to the client to verify the server's authenticity. Since it's a two-way handshake, the server will then request the client certificate to verify the client's authenticity by confirming the certificate against the server's keystore. The keystore is a repository of digital certificates of trusted certificates. After exchanging certificates, the client generates a pre-master secret used for establishing the AES encryption between the two entities. This secret is encrypted with the server's public key. The client then authenticates its own certificate by sending a message to the server signed with its private key to verify its identity. When this is completed, AES encryption can be used between the server and the client, and the connection can be assumed secure.

4.1.1.3 Remote operations

In any server hosting environment, it is essential to be able to operate remotely. In order to lessen the necessity of travel for development and maintenance teams, we employed full remote access to our servers. This was done primarily through remote SSH connections to the servers via Visual Studio Code with the Remote Explorer extension. Visual Studio Code provided a file directory overview, an IDE that we could use to edit code files on the server directly, and the SSH terminal command line on the server was enabled. This greatly increased our efficiency and speed when building our UNS network. The SSH connections are a significant vulnerability if not properly handled. We employed a 20-character password randomized password on the SSH connections for security. For extra measures, we applied a 120-second ban if authentication failed more than four times to reduce the number of brute-force attacks we experienced.

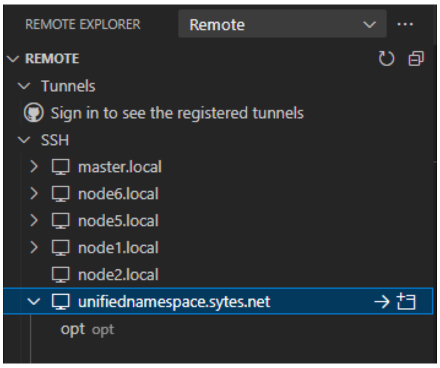


Figure 21 Different remote SSH servers in VSCODE

4.1.1.4 VPN

A VPN entry point was added to the network to further extend our remote operation capability. The VPN enables us to be a participant in the local network. The IPSec VPN is part of the Cisco router and was set up as a host-to-network topology. This security encrypts and authenticates every packet that travels through the network to the remote access VPN access point. Since this entry point of the VPN is from the router, the user has local access to all layer 3 networks.

4.1.1.5 Factory Broker

The factory broker is a standalone broker hosted on an Ubuntu 2022 server located in our server rack at HVL. The factory site broker can host multiple publishers and subscribing SCADA clients. But the maximum number of connections is limited to 25 when using the trial enterprise edition. For all inbound and outbound network access, we have set up a Cisco network, detailed in the Networking chapter.

The clients can connect with the factory broker by setting the correct port of the TCP listeners, further detailed in the Networking chapter. Upon receiving a client, the MQTT broker handles the connection by establishing a secure connection using TLS. After establishing a secure connection, the broker allocates topics for the client according to the client's request. It is possible to limit what topics a client can access, discussed in the Authorization chapter. The broker utilizes the Prometheus extension that allows us to publish all the broker statistics via HTTP for dashboard systems like Grafana to display them.

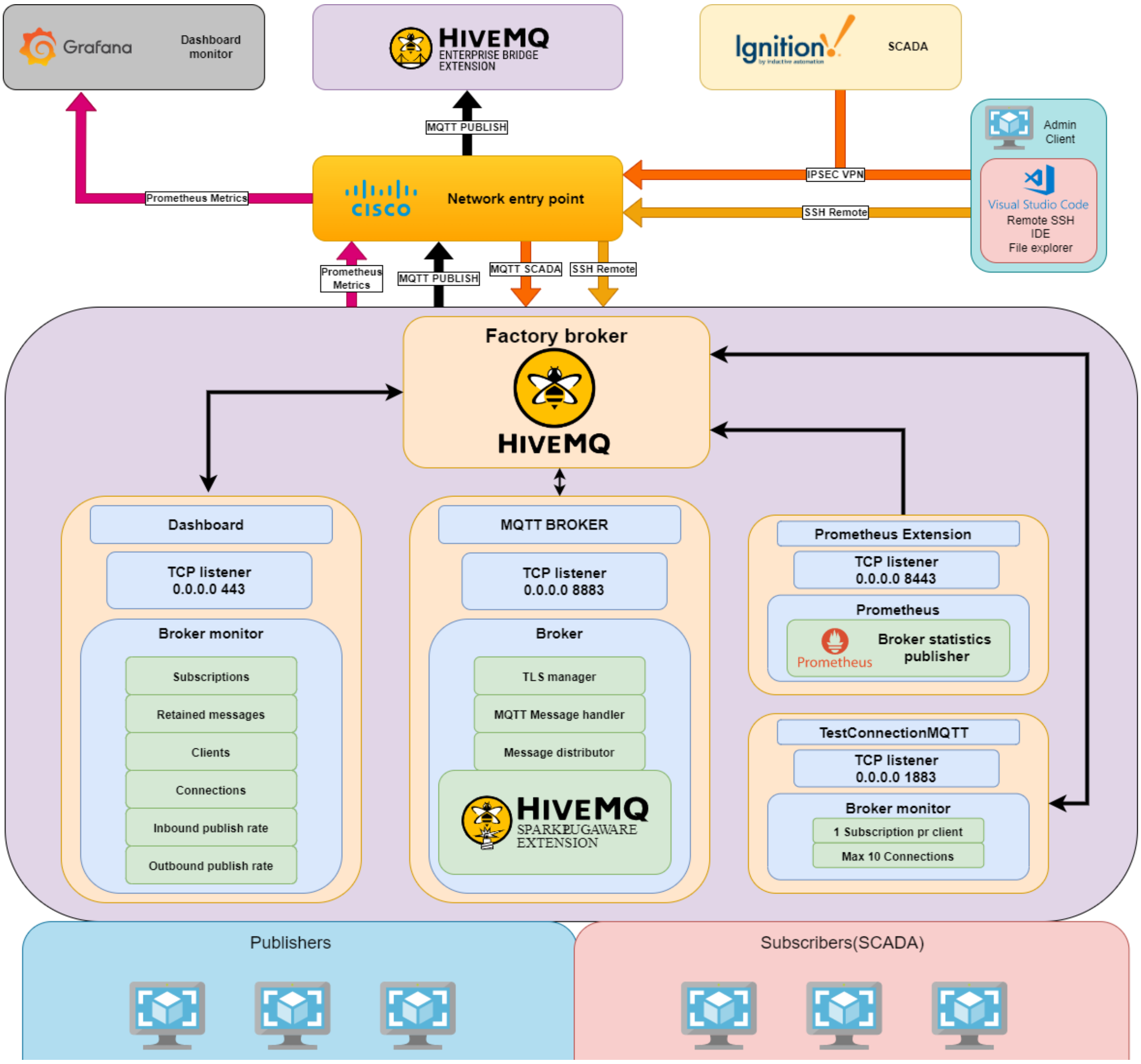
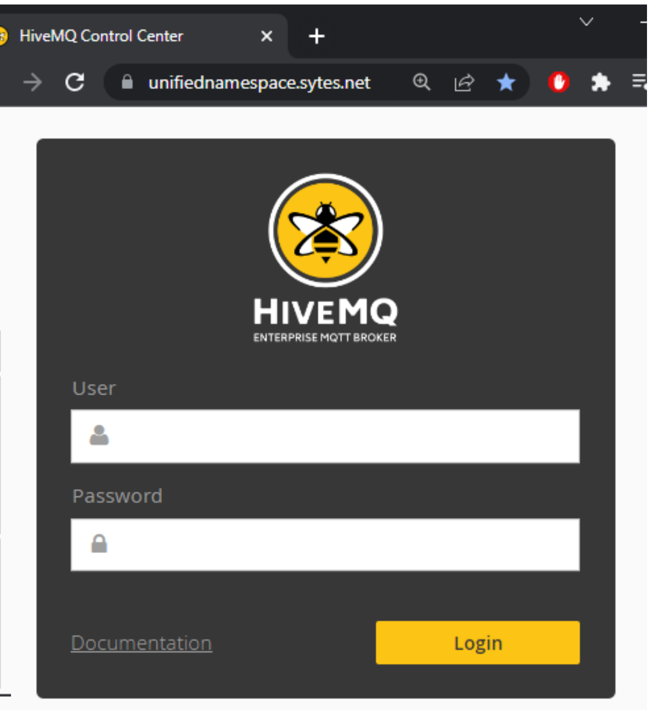


Figure 22 Factory broker and connected component architecture

The factory broker can only handle MQTT traffic from its LAN due to the rules we have enforced in the Cisco network. This is done to block all potential clients from outside the LAN. Consequently, all the topics must be transferred from the factory broker to the cluster broker. This is done by utilizing the HiveMQ bridge extension.

The broker hosts its own website called the dashboard. Here we can monitor what is currently ongoing at the broker, view statistics like available server resources, network traffic, MQTT traffic, connected clients, and more is available through the HiveMQ dashboard. It also allows the user to see retained messages currently on the broker.



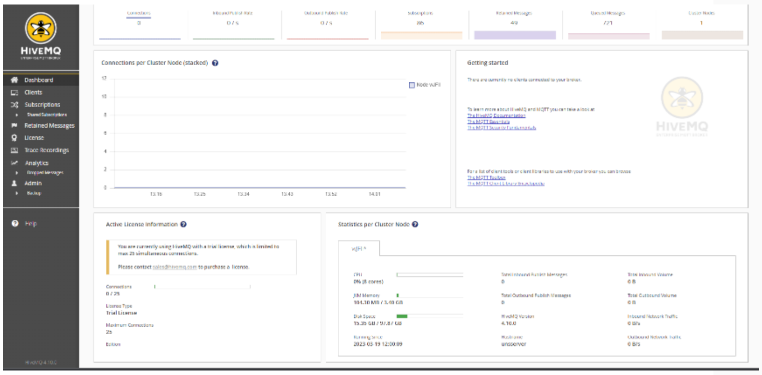


Figure 23 The default HiveMQ (cluster) dashboard

4.1.1.6 Cluster broker

The Cloud broker is the primary broker in the Unified namespace and the broker that holds the entire UNS network. To be able to seamlessly scale the UNS network, the Cloud broker must employ clustering and load-balancing technology to rapidly scale the network as needed. Our implementation will be the production of a deployable cluster image that can be infinitely scaled by deploying more images on standalone servers and applying load-balancing technology.

In this thesis project, we developed two cluster broker solutions. The initial solution was based on Kubernetes and Docker images, which offered high availability and seamless scalability. However, it was discovered that this solution could not be implemented on the Raspberry Pi hardware due to its ARM64 architecture. The project budget constraints also prevented the acquisition of AMD64 servers that could have supported the solution. As a result, a second cluster solution had to be developed specifically for the ARM64 architecture, which led to the loss of Kubernetes and scalability. The limited RAM available on the Raspberry Pi devices further restricted the number of brokers that could be established in the cluster network. Only three were successfully deployed out of five.

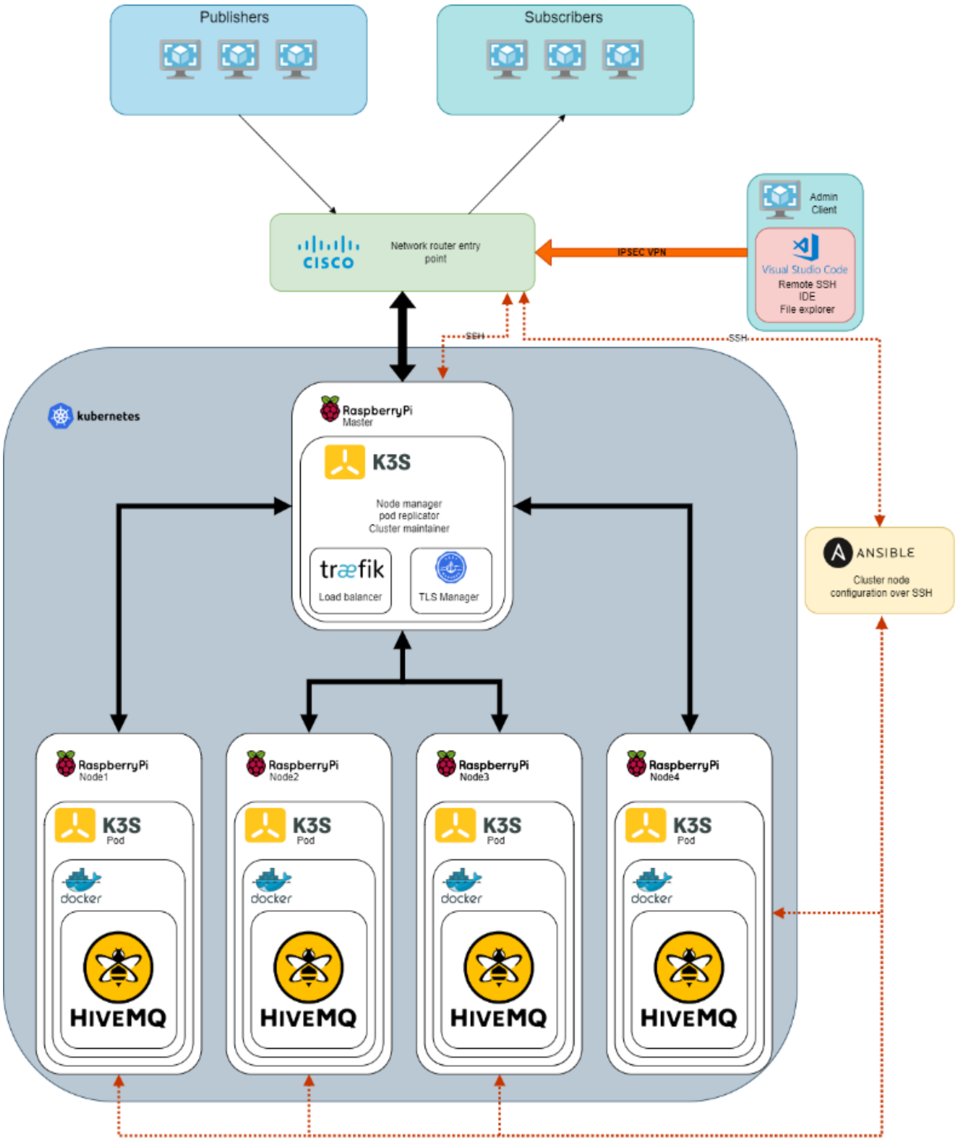


Figure 24 Desired broker cluster implementation using Kubernetes clustering.

4.1.1.6.1 Desired Cluster

Undoubtedly, the best technology to deploy for a cluster solution is Kubernetes, due to its ease of deployment and scalability, as detailed in chapter 3. Initially, we built a cluster using this technology. For the first system, we used K3s software on a Raspberry Pi 64-bit operating system to include Kubernetes. K3s was chosen because it requires the least amount of hardware resources to run Kubernetes. A load balancer with a proxy must be configured on the server to distribute incoming traffic load to the server. K3s comes with an inbuilt load balancer and proxy called Traefik.

The proxy creates a virtual connection point to the cluster by duplicating the Ethernet interface via software. This means, in practice, that a selected server will have two IP addresses. In our case, the chosen master server was assigned the IP address 192.168.10.200, and the proxy was assigned the IP address 192.168.10.210, which served as the entry point for the cluster. All traffic to the cluster was directed from the Cisco router to this IP address. When incoming traffic arrives at the proxy IP address, it enters the load balancer. The load balancer distributes the incoming connections over the connected servers in the cluster by response time. Other parameters can be chosen as distribution parameters, but we opted for response time. After the load balancer selected a server, the connection was sent to that server.

HiveMQ supplied the Docker containers containing the image for the cluster brokers. However, this container was packaged for the amd-64 architecture, rendering it incompatible with the arm-64 architecture on Raspberry Pis. We kindly requested HiveMQ to check if they had an arm-64 image, but unfortunately, they could only advise us to check out their community portal, which did not provide any further insight on how to obtain a potential arm-64 image. This was a major dealbreaker and would have permanently terminated our desired solution. Therefore, we decided to rebuild the supplied amd-64 image. We managed to rebuild the image by manually unpacking it on our Ubuntu server and changing all the dependencies in the package from amd-64 to arm-64 dependencies. I cannot stress enough that this is not a recommended approach. The amount of time used to rebuild this image manually is a luxury only a student may have.

After rebuilding the Docker images, we were able to deploy three HiveMQ brokers to our Kubernetes cluster with automated respawn. This means that if the broker were to crash at any point, the Kubernetes network would shut down the pod containing that image and automatically deploy a new one. This approach also gave us the ability to scale the cluster easily by adding another worker node to the cluster, consisting of a Raspberry Pi with K3s installed. We only needed to change our desired number of brokers from three to four, and Kubernetes would deploy it on the newly installed Raspberry Pi.

The failure of the desired cluster solution was discovered when we attempted to connect clients to the cluster. We found out that even though each individual broker was functional, the brokers themselves did not establish a HiveMQ cluster network to share incoming topics between each other. This meant that we had three individual brokers and not a cluster. After extensive troubleshooting, we discovered that one of the extensions in the rebuilt HiveMQ image was not functioning. This extension's role was to discover and share information between the HiveMQ brokers connected in the network, and it was vital to enabling the HiveMQ broker cluster in Kubernetes. We suspected that the malfunction of this extension was due to our rebuild of the original image, as we were not able to find any entries in the community forum or GitHub with the same error. As this is the only extension that HiveMQ does not share openly on GitHub, we were not able to obtain its source code. No fix was ever found during this thesis to remedy this issue. In order to establish a cluster and build our proof of concept, we had to rethink our solution.

4.1.1.6.2 Final Cluster

After our desired solution failed, we gained a good grasp of the proxies and load balancers technologies. This made it a lot easier to establish a networked broker cluster manually. HiveMQ offers a downloadable extension called HiveMQ discovery. When applied to the servers, this extension allows us to deploy multiple servers in a network, and they will start sharing their content with each other by only supplying the IP address of the individual servers. Our final solution did not contain any Kubernetes, and the broker hosted was a Java virtual machine that hosted a copy of our factory broker with an added cluster and discovery extension from HiveMQ. The disadvantage of this solution was that, in order to scale the network, the Java virtual machine had to be downloaded to the Raspberry Pi and manually added to the Raspberry's services and boot configuration. Since we no longer used the K3s software, we had to obtain new proxy and load balancer software. We chose the HAProxy software since it was an open-source software with minimal configuration needed to host the proxy and load balancer. We used the exact same setup for the proxy and load balancer as our desired solution but now with HAproxy instead of Traefik.

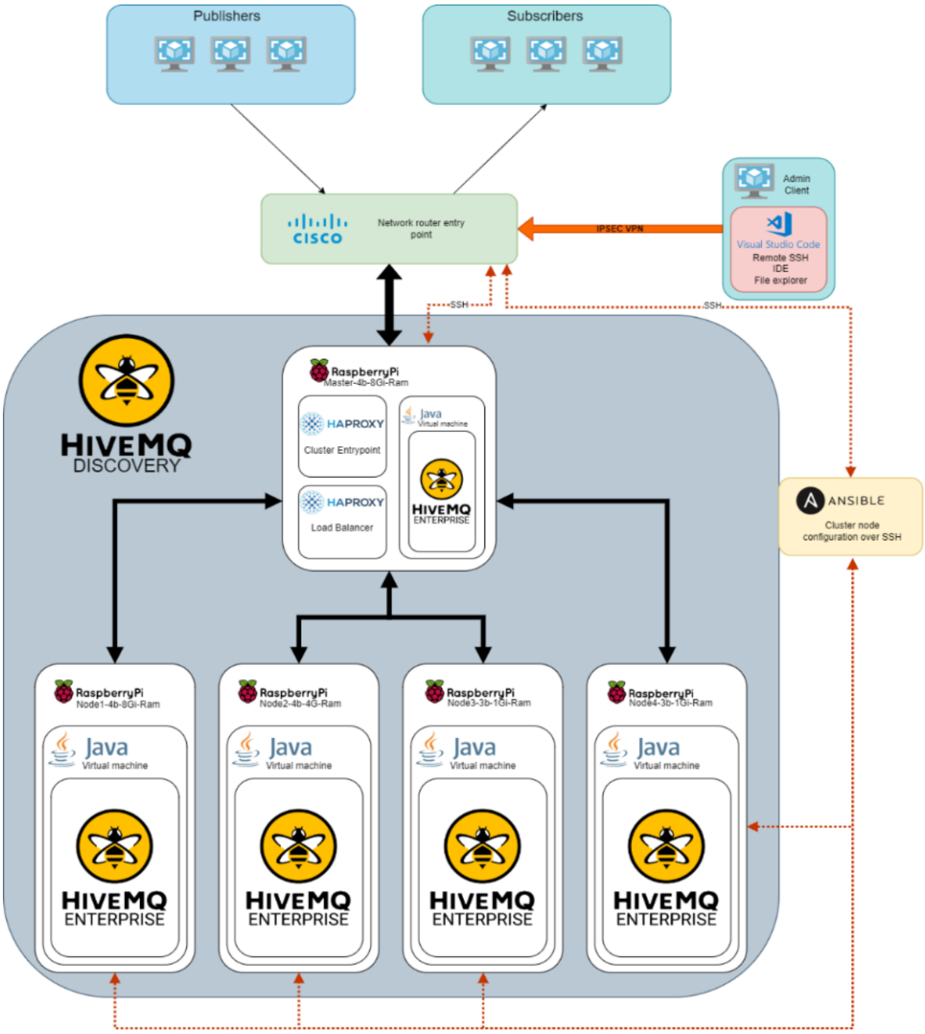


Figure 25 Final broker cluster implementation using HiveMQ extensions.

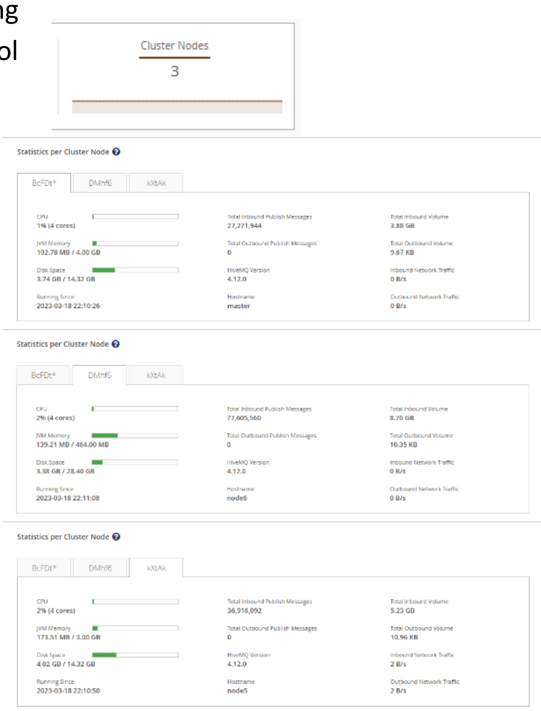
4.1.1.6.3 Control

We wanted to avoid the necessity of configuring each worker node individually. As the cluster grows, the time to configure the worker nodes individually will increase exponentially if not automated. Therefore we have employed Ansible software, a tool used for managing and configuring systems, applications and networks over SSH. This tool is agentless, meaning it does not require any software installed on the remote system.

4.1.1.6.4 Hosting services

During our preliminary research, we decided to use Azure hosting services to host our clustered cloud-broker. During the implementation phase of the cluster-broker in the cloud, financial costs related to hosting services grew dramatically due to the unforeseen and partly hidden costs at Azure hosting. It was therefore decided to build our own cluster for cloud broker implementation.

It was necessary to drop the Ubuntu server OS in the cluster implementation for a Raspbian lite OS which uses fewer resources to build and run a raspberry pi cluster.



Cluster Nodes
3

Statistics per Cluster Node

BcFDt* | DMhf6 | kXtAk

CPU
1% (4 cores)
JVM Memory
192.78 MB / 4.00 GB
Disk Space
3.74 GB / 14.32 GB
Running Since
2023-03-18 22:10:26
Total Inbound Publish Messages
27,271,944
Total Outbound Publish Messages
0
HiveMQ Version
4.12.0
Hostname
master
Total Inbound Volume
3.88 GB
Total Outbound Volume
9.67 KB
Inbound Network Traffic
0 B/s
Outbound Network Traffic
0 B/s

Statistics per Cluster Node

BcFDt* | DMhf6 | kXtAk

CPU
2% (4 cores)
JVM Memory
139.21 MB / 464.00 MB
Disk Space
3.38 GB / 28.40 GB
Running Since
2023-03-18 22:11:08
Total Inbound Publish Messages
77,605,560
Total Outbound Publish Messages
0
HiveMQ Version
4.12.0
Hostname
node6
Total Inbound Volume
8.76 GB
Total Outbound Volume
10.35 KB
Inbound Network Traffic
0 B/s
Outbound Network Traffic
0 B/s

Statistics per Cluster Node

BcFDt* | DMhf6 | kXtAk

CPU
2% (4 cores)
JVM Memory
173.51 MB / 3.00 GB
Disk Space
4.02 GB / 14.32 GB
Running Since
2023-03-18 22:10:50
Total Inbound Publish Messages
36,916,092
Total Outbound Publish Messages
0
HiveMQ Version
4.12.0
Hostname
node5
Total Inbound Volume
5.23 GB
Total Outbound Volume
10.96 KB
Inbound Network Traffic
2 B/s
Outbound Network Traffic
2 B/s

Figure 26 Overview from Cluster dashboard that shows 3 connected servers in the cluster.

4.1.1.6.5 Raspberry PI Cluster

In the initial phase of our Raspberry Pi cluster build, one of the issues that started to arise was the heat. Each of the individual Pis would start reducing their performance due to heat on the boards. It was then decided to build a framework to mount all the Raspberry Pis and add fans to reduce the heat. We designed our own cluster frame and mounts in Fusion360 and 3D printed the frame and brackets. A small perforated board was used to solder eight XHP connectors together to easily connect all the fans to one power source. After the Pis were mounted, no performance drop caused by excessive heat was detected.

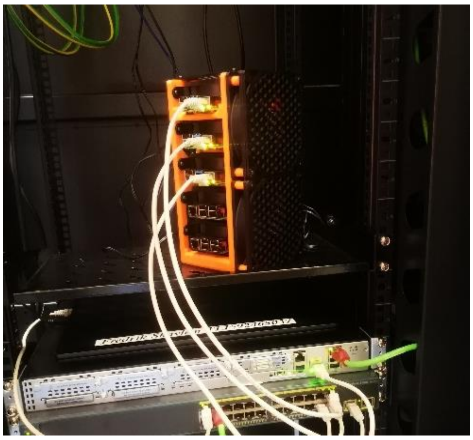


Figure 28 Cluster installed at Hvl site

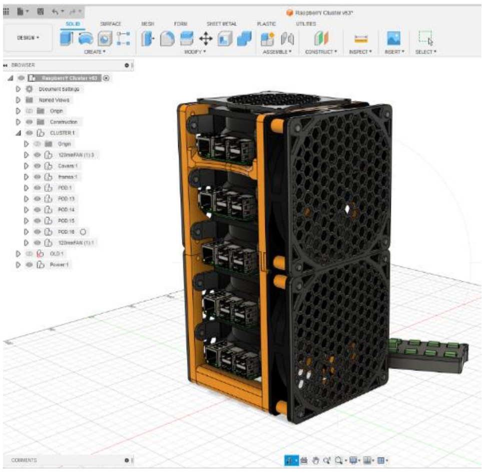


Figure 29 CAD model in Fusion360

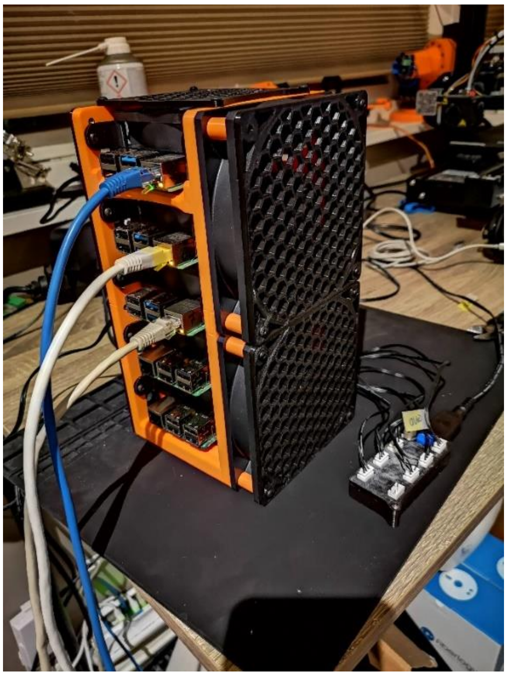


Figure 27 The final build of our raspberry pi cluster

4.1.2 Broker security

The Cluster employs four layers of security. The first layer checks if the connecting IP address is authorized. The connection will be immediately dropped if the inbound traffic IP address is not included in the access list on the router.

The second layer of security is encountered after the connection has passed the access list. Only a handful of routes are available from the router. These routes are statically routed to ensure that no other routes are accessible in the network for inbound and outbound traffic through the router. The same can also be effectively achieved with a firewall.

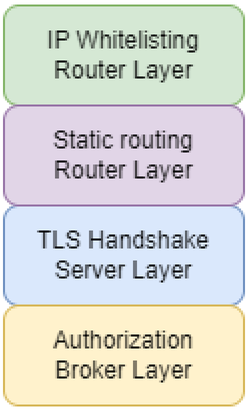


Figure 30 Security layers [1]

The third layer comes into play after the connection is routed to the Cluster on a static port. Here, the inbound connector must establish a two-way TLS connection with the receiving server. If the connector fails to establish this connection, the connection will be terminated.

The last layer restricts what topics a connected user can access after connecting to the broker and is detailed in the Access Control chapter.

4.1.2.1 Access control

When running an extensive network, it's essential to protect the network from unwanted and unintentional user abuse. Situations where one client in the network unintentionally subscribes to all topics in a network with millions of topics will impact the network's performance. The possibility to publish to any topic should also be restricted to prevent inconsistent data or intended manipulation from hosts on the inside publishing to topics owned by others.

Therefore, it's essential to employ some sort of access control mechanism in the network to control the various connecting clients' ability to subscribe and publish topics. HiveMQ has a pre-built extension called enterprise security extension. This extension addresses the issue of access control in the system.

After establishing the TLS connection when connecting to a local or cloud broker, the client must submit a username and password. The authentication manager will first check the username and password to verify if the client is allowed access to the broker. If the client is authenticated, the client is passed on to the Authorization manager. This manager will give the client access to the topics it can view based on the user's role.

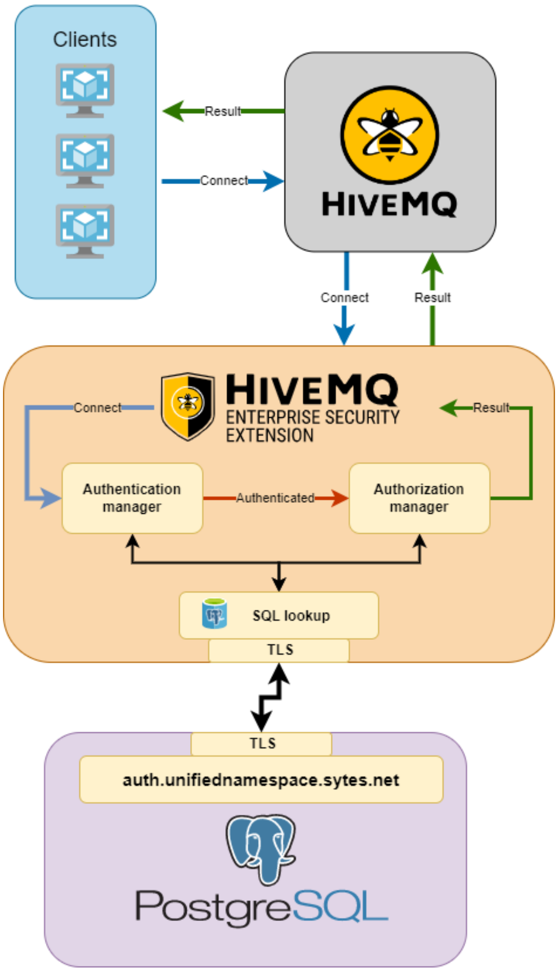


Figure 31 Access control structure

Since the security extension is deployed on all the servers in the network, a centralized user database is needed. We chose to use a PostgreSQL database as the centralized database.

The HiveMQ security extension documentation details how the SQL database should be structured in order to host the user authentication and authorization database. Roles can be set up in the database to make user permissions easier to maintain. These roles are then assigned to the different users by relations in the database. This database is hosted on the cluster under a separate proxy such that it is its own network, detailed as auth server in the network section.

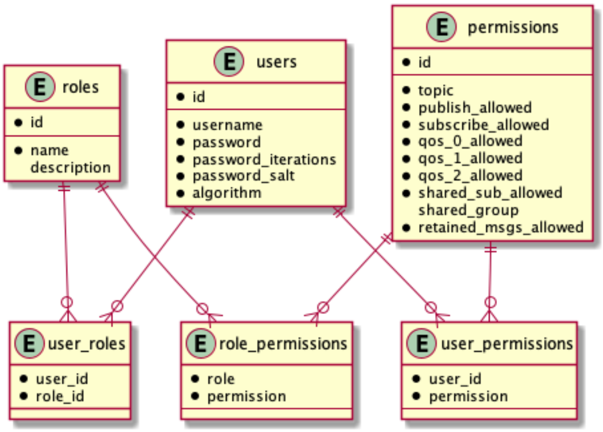


Figure 32 SQL database structure [1]

4.1.3 Network

In any broker solution, the first step is to ensure a viable network solution. We chose to go for a Cisco network solution since it's a widely adopted networking system and partly due to Cisco hardware being readily available. During the thesis, only one global IP address was made available for the project by the IT department at Hvl, so it became necessary to segment the factory broker and cluster broker net to simulate different locations. One of our objectives during the deployment was to force ourselves to utilize remote connections to build and maintain the servers without being at the site to learn how to work in a truly remote environment.

4.1.3.1 Cisco network

Building advanced network solutions with Cisco without using prebuilt configs can lead to long deployment periods and the implementation of unwanted networking relations. In order to mitigate this and to ensure future reconfigurations are getting deployed without any breakage, we build our entire Cisco network solution in Netacad Cisco Packet Tracer. This allows us to test and prove new changes before any deployment to ensure viability.

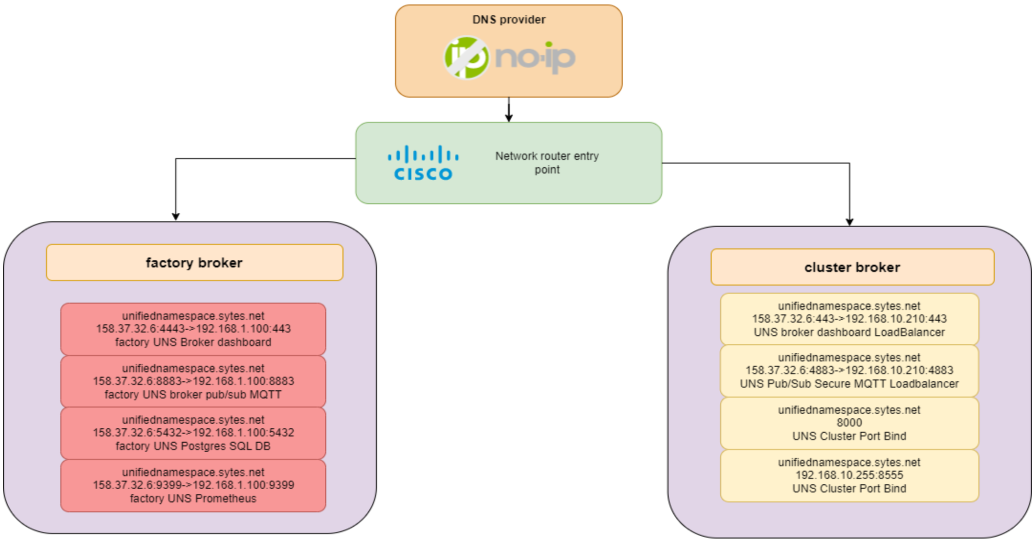

DNS provider
no-ip
cisco
Network router entry point
factory broker
unifiednamespace.sytes.net
158.37.32.6:4443->192.168.1.100:443
factory UNS Broker dashboard
unifiednamespace.sytes.net
158.37.32.6:8883->192.168.1.100:8883
factory UNS broker pub/sub MQTT
unifiednamespace.sytes.net
158.37.32.6:5432->192.168.1.100:5432
factory UNS Postgres SQL DB
unifiednamespace.sytes.net
158.37.32.6:9399->192.168.1.100:9399
factory UNS Prometheus
cluster broker
unifiednamespace.sytes.net
158.37.32.6:443->192.168.10.210:443
UNS broker dashboard LoadBalancer
unifiednamespace.sytes.net
158.37.32.6:4883->192.168.10.210:4883
UNS Pub/Sub Secure MQTT Loadbalancer
unifiednamespace.sytes.net
8000
UNS Cluster Port Bind
unifiednamespace.sytes.net
192.168.10.255:8555
UNS Cluster Port Bind

Figure 33 Port achitecture at HVL server park

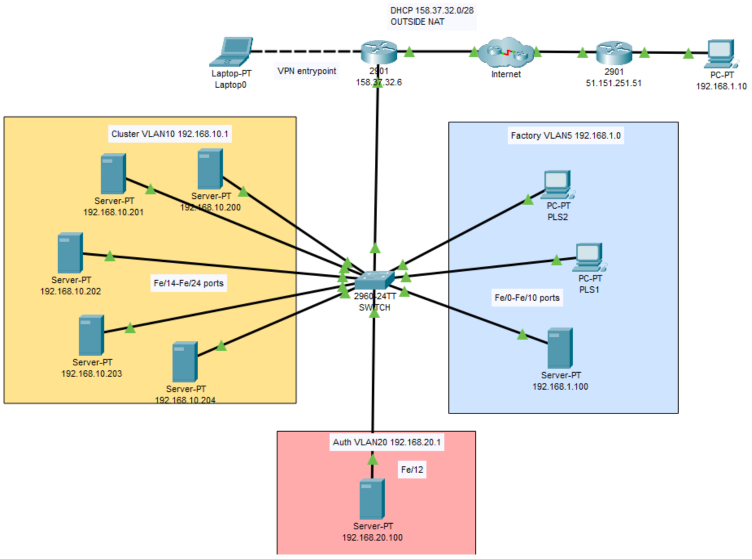


Figure 34 Hvl network simulated in Packet Tracer

Our cisco network contains one Cisco router and one Cisco switch. The router's IP is dynamically allocated from HVL's router using DHCP. It is our global IP entry point to the internet. Our connection to the internet has no preceding port-blocks or firewalls, and is as such an open connection. Meaning in practice that the entire world can access our router, including botnets and malicious actors. To address this, we implemented a whitelist Access-list on the router that filters all inbound traffic and only accepts traffic from the whitelisted IP addresses. The factory broker is allocated the static IP of 192.168.1.100, and the cluster broker entry point has 192.168.10.200. This was done to manually route traffic from the router to the server through static port forwarding on the router.

The deployed router had only two physical ethernet ports. For us to employ three networks on this port, one-to-one links had to be omitted and replaced by a method called trunking. It enables us to convey multiple communications links on one connection. Further, the Cisco router employs one software interface per connected physical ethernet port, but these can be further divided into sub-interfaces. Consequently, the interface going to the local switch was split into three different interfaces acting as particular subnets gateway.

4.1.3.2 Hostname DNS

Using hostname/DNS is essential to not statically commit every device on the network to a single IP address. When a Hostname is used, we can control what IP address is mapped to which hostname, meaning if the IP address were to change, no change in the hostname would occur. During our bachelor thesis, we employed DNS services from a company called No-Ip at https://www.noip.com/.

We initially configured our first server with all devices, including TLS certificates bound to a single IP address. This was a major misstep. In the deployment of our first server, we moved the server location and subsequently changed the server's global IP address. Due to this IP change, we were forced to redo a lot of the code of the connected devices in the network and all TLS keys were voided. After this incident, we adopted the hostname *unifiednamespace.sytes.net* for our server to mitigate this error.

4.1.3.3 Port allocations

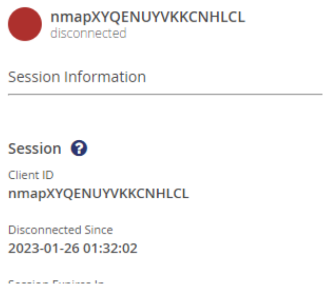
Since both cluster and factory were hosted on the same network, multiple listeners were at the same port, which could have interfered with each other, resulting in clients connecting to the wrong brokers. Therefore, it was necessary to further split the cluster broker and factory broker by ports. We used static routing to separate the different listeners. In practice, if you were to access the dashboard of the cluster, it would be https://unifiednamespace.sytes.net:443/, and if you were to enter the factory dashboard, you would have to change the port to https://unifiednamespace.sytes.net:4443/. The idea was to employ DNS SRV technology to make the port differences more human-readable. The DNS SRV is a DNS service that holds an IP address and a port. e.g. https://unifiednamespace.sytes.net:4443/ would become https://factorydashboard.unifiednamespace.sytes.net and the cluster would become http://clusterdashboard.unifiednamespace.sytes.net. To set up a DNS SRV service at our DNS provider No-ip, we only needed to insert our wanted subdomain of our DNS hostname unifiednamespace.sytes.net with a port number and subdomain name. Accessing this service would cost our project an additional 29.90$, and we decided to keep the ports in the URL and not employ the DNS SRV service.

4.1.3.3.1 TCP listeners

The clients can connect to the MQTT broker simply by using the IP address of the network entry point. The broker holds TCP listeners for the induvial ports such that the broker is listening for connections from potential clients. These listeners can be limited to a range of IP addresses or allow all addresses by setting wildcard IP address 0.0.0.0 and desired port.

4.1.4 Server attacks - challenges of unsecured connections.

This Section describes the server attacks that occurred during our bachelor thesis and is a digression from the report. However, we wanted to add it as an interesting read. On the 26th of January, an intrusion was detected on our hiveMQ broker on the unsecure channel 1883. No IP address was logged. The unknown intrusion subscribed to the topic wildcard # to access all topics. It has been determined that this was just a probe of the system to assess the server as any potential attack target. No additional information was logged due to the server only logging failed connection attempts. This has since been addressed and fixed.



nmapXYQENUYVKKCNHLCL
disconnected

Session Information

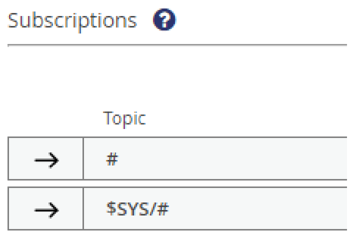
Session

Client ID
nmapXYQENUYVKKCNHLCL

Disconnected Since
2023-01-26 01:32:02

Figure 35 Unknown Foreign Client

This intrusion led to a wider deep dive into our server's authentication logs. It was discovered that between January 22 and January 26, a total of 26023 failed login attempts on the server were detected. 71235 different user and password combinations were tried to primarily access the SSH and CRON port.



Subscriptions

	Topic
→	#
→	$SYS/#

Figure 36 Intruder client subscription

After some analysis of the authentication data logs, it was determined that no particular country of origin could be asserted. Applying whois/whoip on a small set of entries concluded that these attacks originate from no specific country. We found the highest concentration from India, Singapore, Israel, China, Korea, United States, Switzerland, Malaysia and Taiwan. Most of these were dictionary attacks, where the attacker tried different combinations of usernames and passwords until they got disconnected due to too many failed attempts. We believe these attacks are part of automated botnets, randomly scourging through the WAN for entry points.

In the initial days of this project, an informal agreement between us that we would use 20-digit randomized passwords for potential logins connected to WAN was formalized. These attacks legitimacies that decision and reflect that a continued security policy must be maintained.

4.2 Edge devices

This section covers Edge of Network (EON) devices positioned at a network's outermost perimeter, typically where data originates. Examples of edge devices include Programmable Logic Controllers (PLCs), smart sensors, or actuators, all of which have a network connection and the capability to publish data to and subscribe from the broker architecture. Some devices are directly compatible with MQTT, allowing them to be connected directly to the architecture. However, other devices, often referred to as legacy devices, lack this built-in capability and require intermediary devices, known as gateways, which are on the same network and publish and subscribe to data on behalf of the legacy device. An important consideration when these EON devices communicate data is the context specified in the producer to ensure consistency and make it easier for subscribing systems to make sense of the information. Information structuring, often called modeling and added attributes like ranges and engineering units, often called metadata, is a valuable addition to the variables which greatly enhance the data. The following chapter will explore several solutions to showcase the possibilities of connecting data sources to UNS.

4.2.1 Siemens S7-1500 as a data source/producer

We require a device that produces and consumes data to generate information sent or received in the UNS architecture. In order to create a source of information, we have set up a Siemens S7-1500 PLC, an edge device in the UNS architecture. The PLC connects to the physical process, which, in our case, is a lab rig for controlling the water level in a tank. Using the rig, we are simulating a production environment where the information about the real-time status of the process is collected and processed, and control commands and setpoints are sent to actuators to manipulate the process.

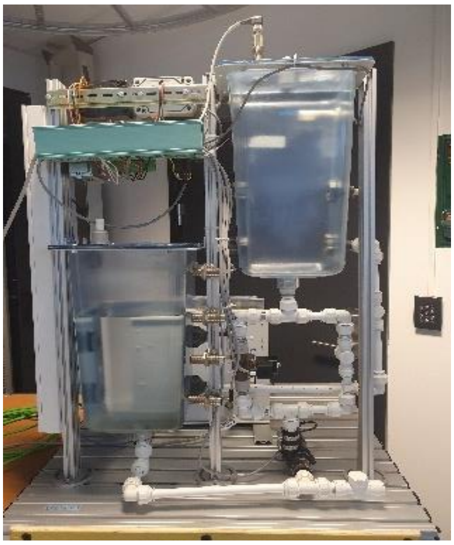


Figure 37 Lab waterrig simulator at HVL

The process is small, and it is therefore easy to keep track of the variables that exist in the namespace. In this isolated case, it only includes our PLC. Since the thesis aims to connect all edge devices in an overall system architecture to make the information available in a common namespace, it is essential to structure information in a way that makes it interpretable and which scales when the system is expanded. Object Oriented Programming (OOP) is a widely used method for structuring and encapsulating logic belonging to a single entity. Objects are often defined based on the physical or abstract reality having functions and attributes necessary to control or gain real-time status of the object. All management occurs through a defined interface that gives the user limited access to manipulate the object. Internal variables are protected, and unnecessary details are abstracted away from the user by hiding them inside the function block. OOP makes it easy to structure variable names to avoid naming clashes and enables unit testing to guarantee that subfunctions work as intended.

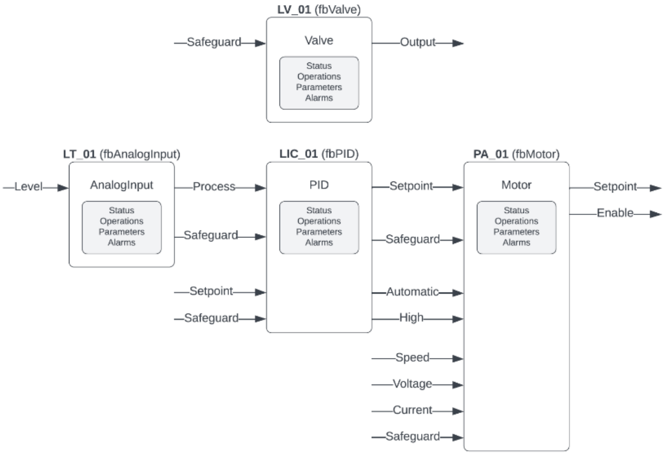


Figure 38 PLC object for controlling the waterrig simulator

Based on the process, we have defined four main objects encapsulating the logic. fbAnalogInput, fbMotor, and fbValve are derived from the physical process. These objects have attributes and functions that should be intuitive for most, even those with no programming background. Functions such as High, Low, Block, and Suppress are all defined in the NORSOK standard. *"The NORSOK standards are developed by the Norwegian petroleum industry to ensure adequate safety, value adding and cost effectiveness for petroleum industry developments and operations." [31, 32]*. The fourth object fbPID is an abstract object containing the algorithm that calculates the control effort based on the error between the process value and the desired setpoint. It also follows the same convention having the same attributes as the NORSOK standard.

The most commonly used functions defined in the NORSOK standard used in the project are:

- **High**: Switch equipment to high, running, open position or feedback indicating the same states
- **Low**: Switch equipment to low, stopped, closed position or feedback indicating the same states
- **Auto**: The object is controlled by logic in the programmable controller
- **Manual**: The object is controlled by commands from Hmi or other network interfaces
- **Suppress**: Object alarms will not be visible to the operator. No actions triggered by alarms will be executed.
- **Block**: Safeguarding of the object is disabled. Alarm annotation is not affected.
- **Safeguard**: Actions to prevent the object from entering a dangerous state or causing harm.
- **Lock**: The object state is changed permanently. New command required to change the object state
- **Force**: Object state is changed as long as force input is high, then returns to the previous state

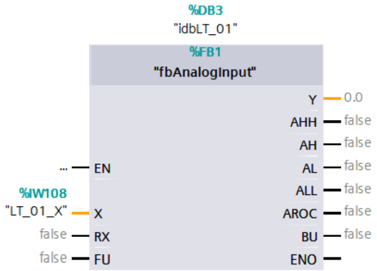


Figure 39 fbAnalog function block

A level transmitter (LT_01) mounted above the reservoir measures the level inside the control tank. An action alarm for high-high level will interlock the pump to avoid further increasing the water level. Several high and low limits will trigger additional alarms with no actions. Alarms can be suppressed to disable the safeguarding caused by HH alarm.

A PID controller (LIC_01) calculates the control signal to maintain desired water level. Control algorithm parameters are provided by the network interface. When operating in automatic mode, the pump's control signal is calculated based on the error between the measured water level and an AutomaticSetpoint. A fixed ManualSetpoint provided through the network interface will determine the desired pump speed when in manual mode. Setpoint tracking has been implemented to force the inactive setpoint to follow so that bumpless transfer is achieved when switching operating modes. A low-level proximity switch in the reservoir will interlock the controller in the low state to prevent the pump from running dry. The controller is locked in manual mode when interlocked. Controller alarms can be suppressed, and interlocking can be blocked by object commands.

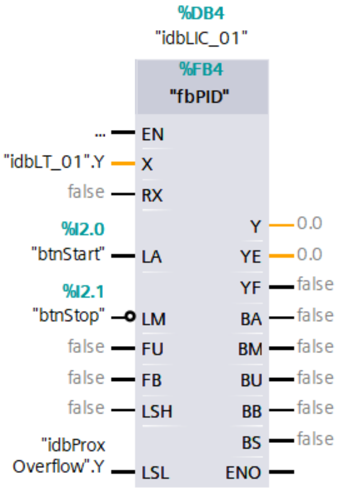


Figure 40 fbPID finction block

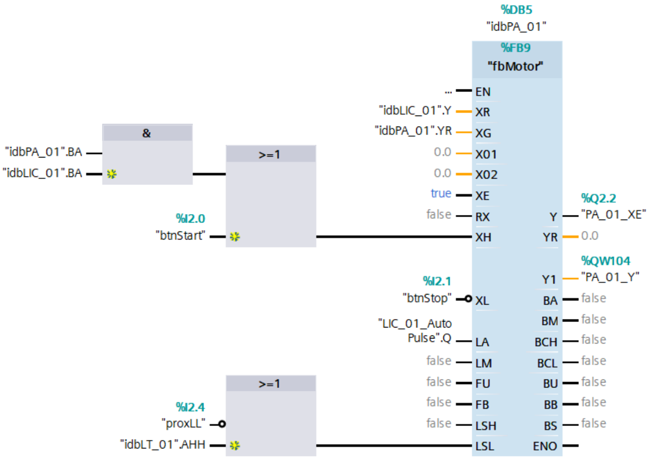
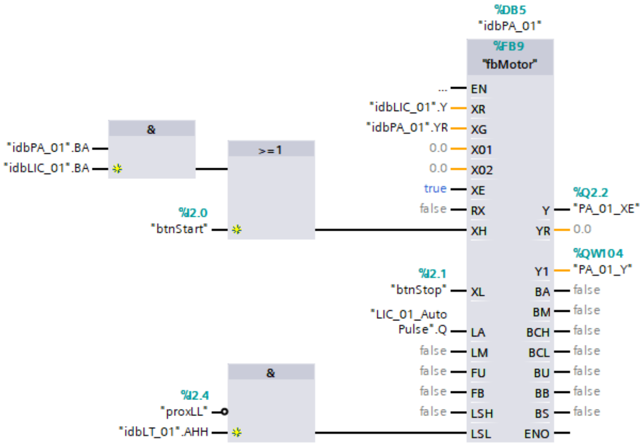


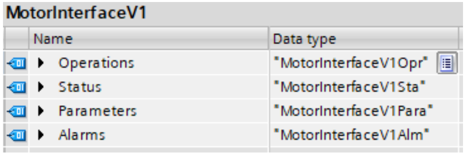
Figure 41 fbMotor function block

The pump (PA_01) will be switched to automatic mode when the controller transitions to automatic mode. Speed reference is received from the controller when operating in automatic mode. A fixed speed can be set through the network interface in manual mode. The motor is interlocked by a high-high water level alarm in the control tank and by the low-level proximity switch in the reservoir. The block has inputs for motor speed, current, and voltage which are evaluated based on alarm conditions and transferred to the control interface to provide real-time status of the pump. Alarms can be suppressed, and interlockings can be blocked. The outputs from the block are connected to the PLC hardware for controlling the pump.

A valve to adjust the pump resistance can only be controlled in manual mode. State and feedback alarms are evaluated, alarms can be suppressed, and are transferred to the network interface. The block has functionality for interlocking and blocking, which is not in use in our implementation.

The interface of these objects is separated into two parts. The object will respond to commands from both interfaces based on the mode of operation. It has one part that is exposed as inputs and outputs of the block. These attributes are connected logically in the program for automatic control by the PLC. The other part of the interface is exposed as a nested data structure grouping different attributes based on the type of function. It is for manual control of the object and is available over the network for operation by Human Machine Interface (HMI) terminals or through an overarching SCADA system, possibly connected through a UNS architecture. Grouping inside the structures simplifies integrations for developers because it adds context to the data. Both parts of the interface can change between Automatic or Manual mode.



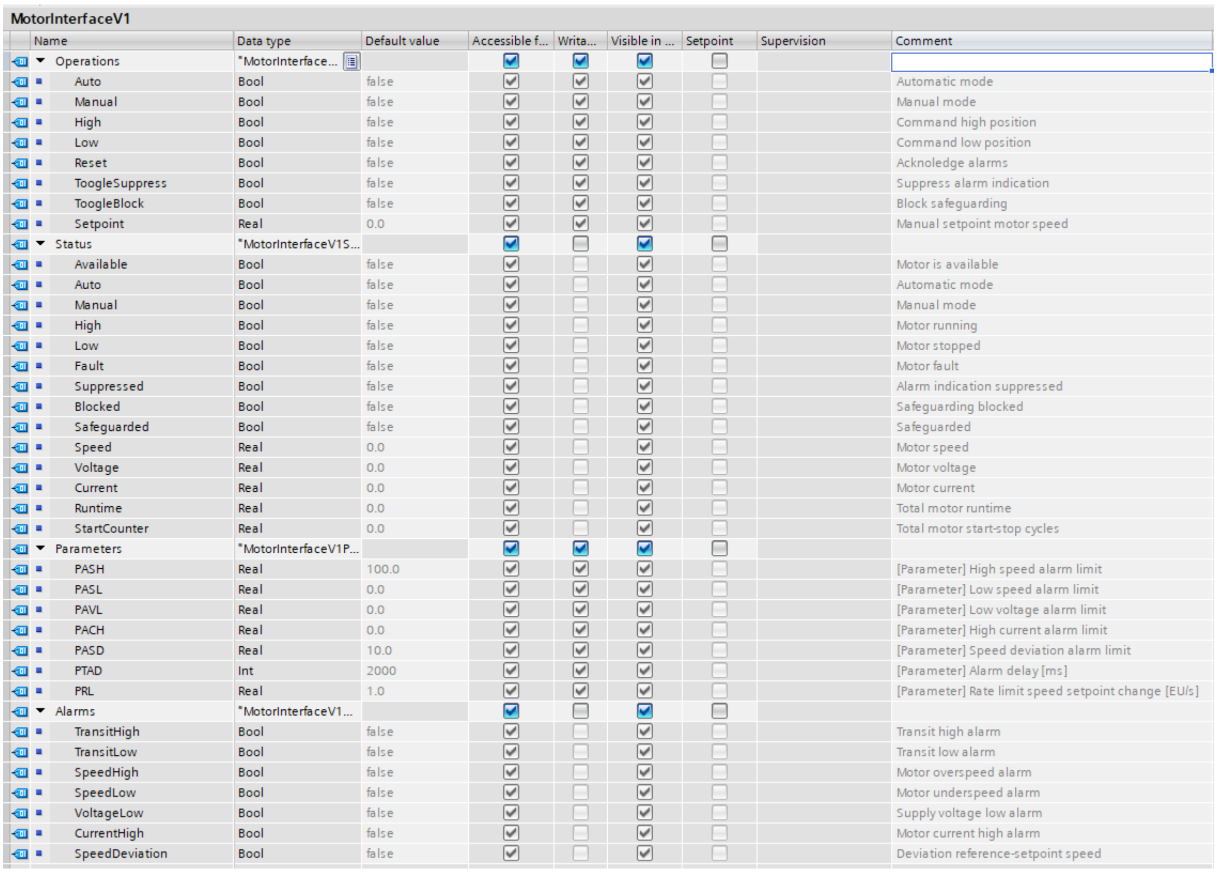


MotorInterfaceV1

Name	Data type
Operations	"MotorInterfaceV1Opr"
Status	"MotorInterfaceV1Sta"
Parameters	"MotorInterfaceV1Para"
Alarms	"MotorInterfaceV1Alm"

Figure 43 Object internal interface

Figure 42 Object network interface



MotorInterfaceV1

Name	Data type	Default value	Accessible f...	Writa...	Visible in ...	Setpoint	Supervision	Comment
Operations	"MotorInterface...		☑	☑	☑	☐		
Auto	Bool	false	☑	☑	☑	☐		Automatic mode
Manual	Bool	false	☑	☑	☑	☐		Manual mode
High	Bool	false	☑	☑	☑	☐		Command high position
Low	Bool	false	☑	☑	☑	☐		Command low position
Reset	Bool	false	☑	☑	☑	☐		Acknoledge alarms
ToogleSuppress	Bool	false	☑	☑	☑	☐		Suppress alarm indication
ToogleBlock	Bool	false	☑	☑	☑	☐		Block safeguarding
Setpoint	Real	0.0	☑	☑	☑	☐		Manual setpoint motor speed
Status	"MotorInterfaceV1S...		☑	☐	☑	☐		
Available	Bool	false	☑	☐	☑	☐		Motor is available
Auto	Bool	false	☑	☐	☑	☐		Automatic mode
Manual	Bool	false	☑	☐	☑	☐		Manual mode
High	Bool	false	☑	☐	☑	☐		Motor running
Low	Bool	false	☑	☐	☑	☐		Motor stopped
Fault	Bool	false	☑	☐	☑	☐		Motor fault
Suppressed	Bool	false	☑	☐	☑	☐		Alarm indication suppressed
Blocked	Bool	false	☑	☐	☑	☐		Safeguarding blocked
Safeguarded	Bool	false	☑	☐	☑	☐		Safeguarded
Speed	Real	0.0	☑	☐	☑	☐		Motor speed
Voltage	Real	0.0	☑	☐	☑	☐		Motor voltage
Current	Real	0.0	☑	☐	☑	☐		Motor current
Runtime	Real	0.0	☑	☐	☑	☐		Total motor runtime
StartCounter	Real	0.0	☑	☐	☑	☐		Total motor start-stop cycles
Parameters	"MotorInterfaceV1P...		☑	☑	☑	☐		
PASH	Real	100.0	☑	☑	☑	☐		[Parameter] High speed alarm limit
PASL	Real	0.0	☑	☑	☑	☐		[Parameter] Low speed alarm limit
PAVL	Real	0.0	☑	☑	☑	☐		[Parameter] Low voltage alarm limit
PACH	Real	0.0	☑	☑	☑	☐		[Parameter] High current alarm limit
PASD	Real	10.0	☑	☑	☑	☐		[Parameter] Speed deviation alarm limit
PTAD	Int	2000	☑	☑	☑	☐		[Parameter] Alarm delay [ms]
PRL	Real	1.0	☑	☑	☑	☐		[Parameter] Rate limit speed setpoint change [EU/s]
Alarms	"MotorInterfaceV1...		☑	☐	☑	☐		
TransitHigh	Bool	false	☑	☐	☑	☐		Transit high alarm
TransitLow	Bool	false	☑	☐	☑	☐		Transit low alarm
SpeedHigh	Bool	false	☑	☐	☑	☐		Motor overspeed alarm
SpeedLow	Bool	false	☑	☐	☑	☐		Motor underspeed alarm
VoltageLow	Bool	false	☑	☐	☑	☐		Supply voltage low alarm
CurrentHigh	Bool	false	☑	☐	☑	☐		Motor current high alarm
SpeedDeviation	Bool	false	☑	☐	☑	☐		Deviation reference-setpoint speed

Figure 44 Expanded network interface of fbMotor object

The expanded view of a motor object's variables demonstrates how variables are organized into a nested structure for simplified grouping based on function type. The same structure can be implemented on SCADA objects described later for simplified interoperability and reusability. Four standard groups defined for all objects are:

- **Operations** contain commands that can be sent to the object to make it change state or operate differently.
- **Status** is feedback from the object revealing the real-time state, often displayed on SCADA systems.
- **Parameters** enable the same class to be instantiated several times with instance-specific configurations like a range or alarm limit specific to the actual object implemented.
- **Alarms** triggered by the object can be used to execute a preventive action inside the PLC logic or relayed to SCADA systems for operator notification.

4.2.2 OPC UA server

Not all devices are compatible with MQTT but offers other alternatives for communication. One of the most used standards to exchange industrial data between devices is OPC UA. Most vendors offer OPC UA servers and clients as an integral part of the programmable controllers or as separate add-on interface modules. It enables system designers to expose internal PLC variables to any OPC UA compatible device connected to this endpoint. In our thesis, we have chosen to set up an OPC UA server on the Siemens S7-1500 PLC to share the PLC variables used by the control system. Configuration of the server, setting up a certificate management system, as well as structuring of data on the server was essential to get in place so that other systems could efficiently access the process data through the UNS. A conceptual design will be described here, and a more detailed implementation can be seen in the appendix.

An essential part of setting up the OPC UA server is configuring the endpoint to which clients will connect. This involves configuring TCP/IP, Security, and Namespaces settings. An endpoint can have several different security levels, but a good practice is to turn off everything except for Basic256Sha256, which is a level that most devices support.

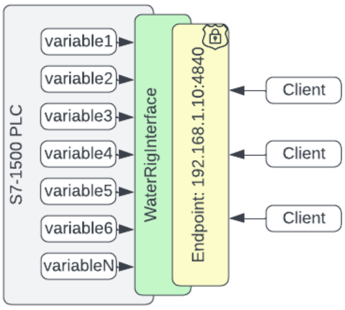


Figure 45 OPC UA server endpoint

OPC UA is based on defining Namespaces where variables are assigned identifiers for addressing. In our setup, we have created a separate namespace called WaterRigInterface with Id ns=4 to better organize the variables and which becomes part of the variable addressing. The full variable address also includes a Node Id, a string identifier or sequentially assigned integer by the PLC, and is unique within each namespace. An example of a complete node address is then: ‘ns=4,i=56’. The OPC standard prioritizes the user being online and browsing the address space since these names are not very intuitive.

TIA portal software from Siemens can function as a certificate authority, creating and signing device certificates. This can be quite practical, as only the TIA CA root certificate needs to be exported to MQTT brokers. Then all devices, such as PLCs, that have had their certificate generated and signed by TIA portal will automatically be trusted by the brokers with no additional requirements.

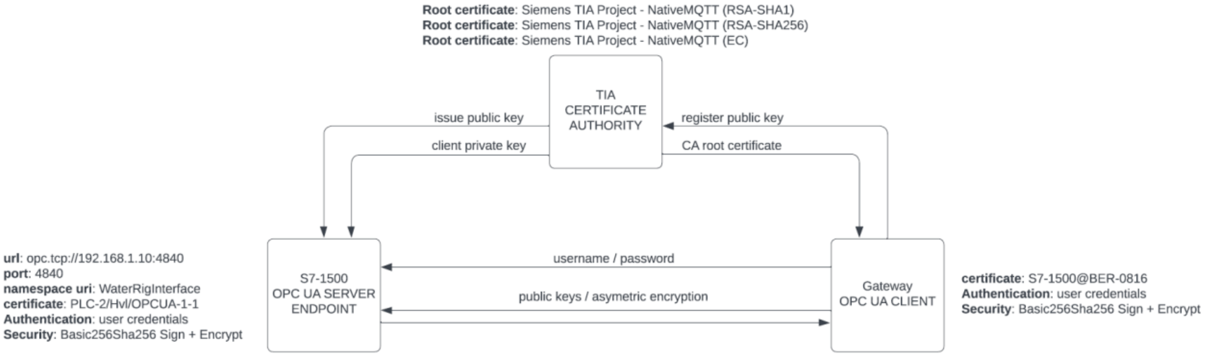

Root certificate: Siemens TIA Project - NativeMQTT (RSA-SHA1)
Root certificate: Siemens TIA Project - NativeMQTT (RSA-SHA256)
Root certificate: Siemens TIA Project - NativeMQTT (EC)
TIA CERTIFICATE AUTHORITY
issue public key
client private key
register public key
CA root certificate
S7-1500 OPC UA SERVER ENDPOINT
Gateway OPC UA CLIENT
username / password
public keys / asymetric encryption
url: opc.tcp://192.168.1.10:4840
port: 4840
namespace uri: WaterRigInterface
certificate: PLC-2/Hvl/OPCUA-1-1
Authentication: user credentials
Security: Basic256Sha256 Sign + Encrypt
certificate: S7-1500@BER-0816
Authentication: user credentials
Security: Basic256Sha256 Sign + Encrypt

Figure 46 TIA portal functioning as a certificate authority

OPC UA is the solution in many industrial plants being regularly used for device-to-device and SCADA communication, thereby having many followers. But it has some weaknesses in that it is a complicated and comprehensive standard, consequently not being optimal for the UNS architectures. Additionally, it is not compatible with the MQTT protocol. In the following chapters, we will explore possible solutions to achieve the unified architecture we seek.

4.2.3 MQTT regular communication

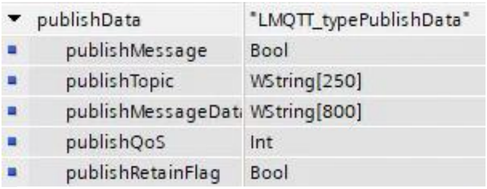
Directly publishing data from the device which produced it is the desired method for achieving a decoupled system architecture. Newer PLCs from Siemens include MQTT drivers, and distributed communication libraries can be added to the project for handling the MQTT broker connection. The library includes a client as a Function Block (FB) for publishing or subscribing to regular MQTT messages.

4.2.3.1 Communication principle

The client is based on a User Defined datatype (UDT) that includes fields for the various MQTT parameters determining the session characteristics. Messages can be published or subscribed to by executing the client FB after configuring the UDT and by setting the block input parameters accordingly. TLS can be selected using the certificates imported into the TIA portal certificate store.

4.2.3.2 Structure of information

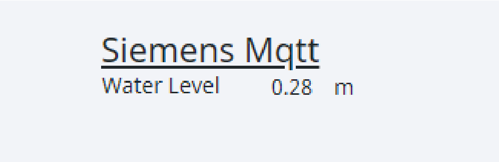
MQTT is payload agonistic, having no restrictions on the content inside messages. Depending on the type of client FB used, it is possible to provide the message content as either a string or a byte array, meaning that the content must be serialized. Functions to convert data types into strings are included in the standard library. Alternatively, project function blocks can be created for serialization specific to the data destined to be transmitted. The encoding of messages is not optimal and unsuited for sending extensive amounts of data. Still, it could be an alternative for sending or receiving a string containing a fixed set of monitored values or less frequent commands between the PLC and a selected communication partner.



▼ publishData	"LMQTT_typePublishData"
▪ publishMessage	Bool
▪ publishTopic	WString[250]
▪ publishMessageDat	WString[800]
▪ publishQoS	Int
▪ publishRetainFlag	Bool

Figure 47 Part of Siemens Mqtt client UDT

To demonstrate the possibility of sending MQTT messages directly from the PLC, we have serialized the water level value into four UTF bytes and placed them into a pre-made byte array in the UDT used by the MQTT client. A periodic task in the PLC operating system triggers the client Function Block every 1000ms, which then transmits the message. The message {"waterlevel":XXXX,"units":"m"} is then transmitted and received by the Engine module described in further detail later in the report. The value is then available on the SCADA system host and is displayed on the HMI screen.



Siemens Mqtt

Water Level 0.28 m

Figure 48 Waterlevel received by Ignition

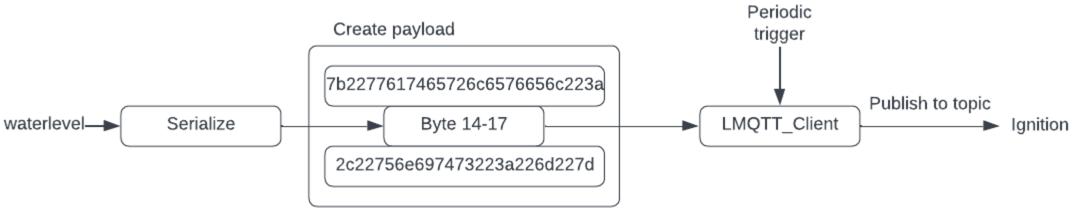


Figure 49 Signal flow of transmitting MQTT messages from Siemens PLC to UNS

4.2.4 OPC UA PubSub

OPC UA is not a protocol but a framework of protocols such as MQTT which enables communication between devices. PubSub with the option of MQTT as the transport protocol is an option in the OPC UA specification to send messages only on change and with a one-to-many principle. Other options like UDP/IP also exist. The communications library includes FBs for publishing or subscribing messages in different encodings to topics in the UNS architecture. It is thus possible to send data using the OPC UA PubSub framework without intermediary gateways which are explained in the next chapter.

4.2.4.1 Communication principle

The PubSub library is based on UDTs for storing connection parameters and handling variables published or subscribed over the MQTT architecture. Data transmitted is divided into WriterGroups, which in turn are divided into DataSetWriters with associated PublishedDataSets. Executing the LOpcUa_PubMqttxxxx function block included in the communication library will publish the variables in the UDT specified through the input to the FB. Transmission can be either periodically or based on an event trigger programmed in the PLC's logic.

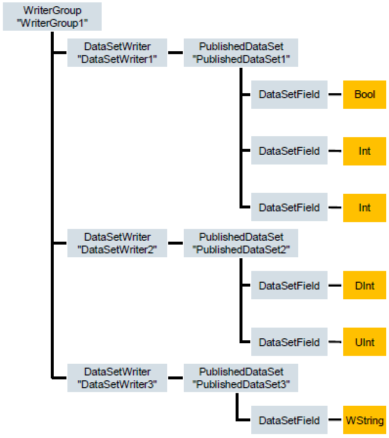

WriterGroup "WriterGroup1"
DataSetWriter "DataSetWriter1"
PublishedDataSet "PublishedDataSet1"
DataSetField
Bool
DataSetField
Int
DataSetField
Int
DataSetWriter "DataSetWriter2"
PublishedDataSet "PublishedDataSet2"
DataSetField
DInt
DataSetField
UInt
DataSetWriter "DataSetWriter3"
PublishedDataSet "PublishedDataSet3"
DataSetField
WString

Figure 50 PubSub data organization [4]

4.2.4.2 Structure of information

Variables must be written to and read out of DataSetFields in the UDTs that the PubSub library utilizes. Only basic data types like boolean, int, real, etc., can be assigned to Datafields. Consequently, complex structures must be flattened into 1D lists, losing part of the context, before publishing the entire DataSetWriter. Allocating one PublishedDataSet for each complex model or variable group like “Commands” or “Parameters” is recommended to maintain data association and context as much as possible.

As a proof of concept, we have implemented the PubSub client function block in the Siemens PLC and successfully connected it to the MQTT broker. We have verified that messages were sent and could be subscribed to by any MQTT client. The structure inside the messages is structurally different and intended to be received by another PubSub client. Still, Json is one of several possible alternatives for encoding and makes it possible to parse out values. The procedure for sending is nearly identical to regular MQTT, except that values do not need to be manually converted into bytes before sending.

PubSub works for communication over MQTT, but it requires cooperation and agreement between the parties setting up both sides of the communication. This is because the context and structures are not yet intuitive or optimized according to the UNS principle.

4.2.5 OPC UA software gateway

Before a common protocol stack has been agreed upon, one solution is to write protocol translators, often called gateways, which receive data on an interface and then format it before sending it to the unified namespace architecture. This solution makes it possible to design the UNS without considerations for legacy systems that do not natively support the desired protocol stack.

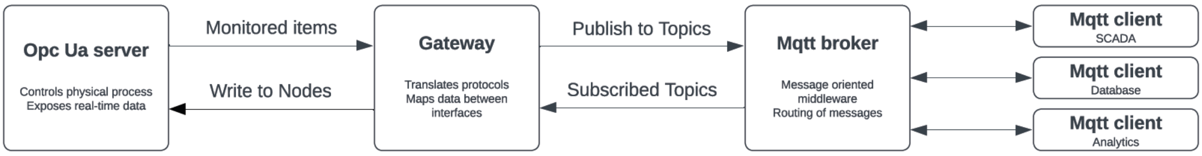
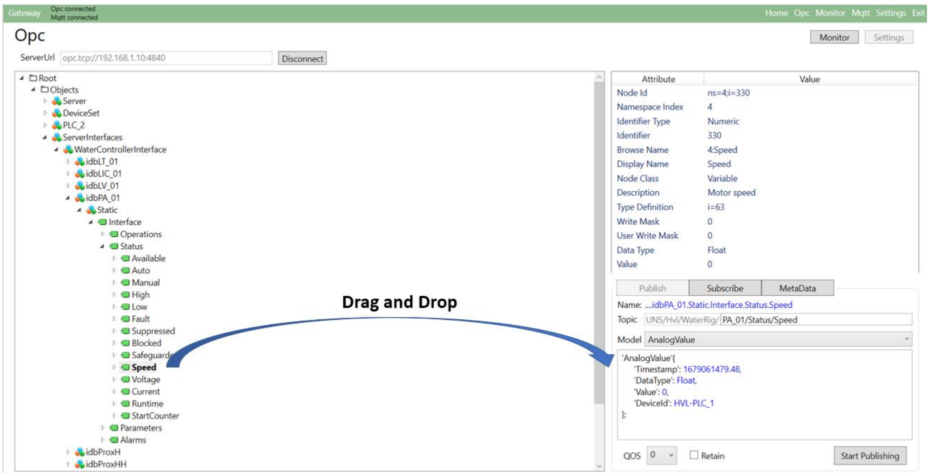


Figure 51 Principal flow of information from the edge PLC to UNS

In this thesis, we have decided to write a gateway application that receives data on an OPC interface and publishes it to UNS topics to better understand the implications of combining different technologies. It is not comprehensive regarding information modeling, complex structures or property bindings but should help understand problems and the various strengths and weaknesses encountered when implementing a UNS architecture. Although it is not the recommended solution to write a protocol translator, it has given a lot of valuable insight into the issues such a device must consider. It should be the job of companies that can maintain such a product over time to build an application with all the necessary functionality that a comprehensive information gateway must be able to perform and which has the required robustness to ensure real-time communication and reliability.

We have intended to make the user interface as intuitive as possible, so configuring the application would only require minimal training. The user interaction aims to ensure that data sent to the UNS is structured in a recognizable format and has sufficient data context to be valuable to data consumers that subscribe to it through topics in the UNS. The only requirement for the user should be to know the address of the OPC server endpoint and the MQTT broker, and to possess authentication credentials to establish a secure connection to these endpoints. After the connection is set up, most of the configuration will be done online by browsing the server address space and navigating through the application User Interface (UI). There shall be no requirement that the user has detailed prior knowledge of the structure or type of data before configuration is carried out. Drag and drop is widely used to improve the user experience and is implemented to map data between the interfaces.



ServerUrl opc.tcp://192.168.1.10:4840

Attribute	Value
Node Id	ns=4;i=330
Namespace Index	4
Identifier Type	Numeric
Identifier	330
Browse Name	4:Speed
Display Name	Speed
Node Class	Variable
Description	Motor speed
Type Definition	i=63
Write Mask	0
User Write Mask	0
Data Type	Float
Value	0

Name: ....idbPA_01.Static.Interface.Status.Speed

Topic UNS/Hvl/WaterRig/ PA_01/Status/Speed

Model AnalogValue

```
'AnalogValue'{
    'Timestamp': 1679061479.48,
    'DataType': Float,
    'Value': 0,
    'DeviceId': HVL-PLC_1
};
```

QOS 0 □ Retain

Figure 52 Online configuration and drag&drop to configure information sharing in gateway.

Only a conceptual description of the gateway application will be given in the main part of the thesis. It is intended to better understand a gateway's function in the overall system architecture. The source code and a detailed description of the application structure, including design choices, have been attached in the Appendix F for readers interested in further and more advanced details.

4.2.5.1 Overall application structure

The application is built with a service-oriented architecture enabling it to connect to any OPC UA server or MQTT broker implementing the specifications. Roughly, it can be divided into three main categories based on what function is to be solved by the specific classes.

The application's fundamental functions rely on data access implemented with the classes OpcDataAccess, MqttDataAccess, and OpcToMqttMapperService, which define methods for establishing sessions, handling service requests inside the sessions and keeping a record of communication statistics. Extensive use of services defined in the NetStandard.Opc.Ua and M2MqttDotnetCore library is used to perform these functions. The classes are independent of a graphical UI and can easily be imported as a library to a console or web application. OpcDataAccess and MqttDataAccess are implemented with interfaces that define what functionality outside classes can request. Internal logic for handling the communication to the remote endpoint is protected. Replacing these classes with equal ones that implement the same interfaces is simple since all functionality is guaranteed through the contract stated by the interfaces.

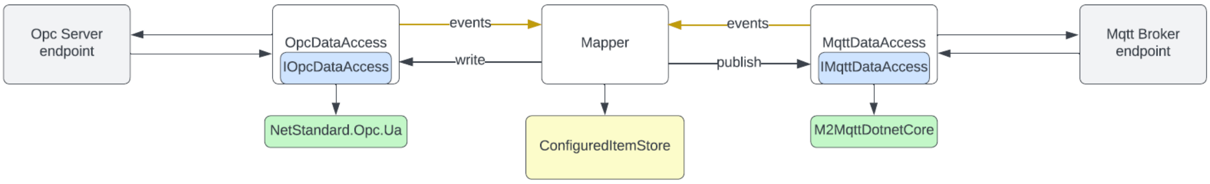


Figure 53 OPC Gateway classes used to handle data access to remote endpoints

Fundamental to the application is the handling of information passed between the two interfaces. Data structures have been designed to organize and ensure data context. A requirement for these structures is that data lookup and transfer must be optimized to guarantee minimal latency and maximum capacity. Collections implementing hash keys are preferred since they guarantee unique keys and efficient searching. Hash lookup is on the order of $O(1)$. The application can select between Json, the most used encoding for data transfer, and Protobuf, a promising encoding scheme developed by Google that guarantees a predictable data structure and is optimized for size. Templates for InformationModels have been designed to ensure that required data attributes are included and that the information content is predictable to the receiver. These templates are separated into a class library for easy maintenance and decoupling from the application logic. A drawback of MQTT is that datatypes are not defined. OPC and PLC are dependent on predefined types. DataType is therefore a mandatory attribute to all InformationModels. The OPC UA definitions for data types are used as a basis.

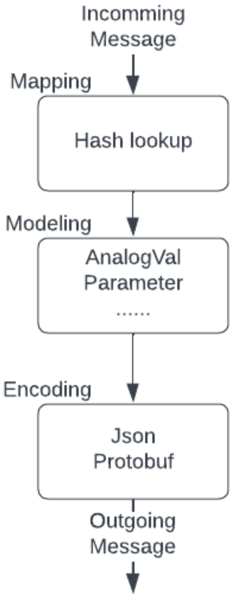


Figure 54 Information flow through OPC Gateway

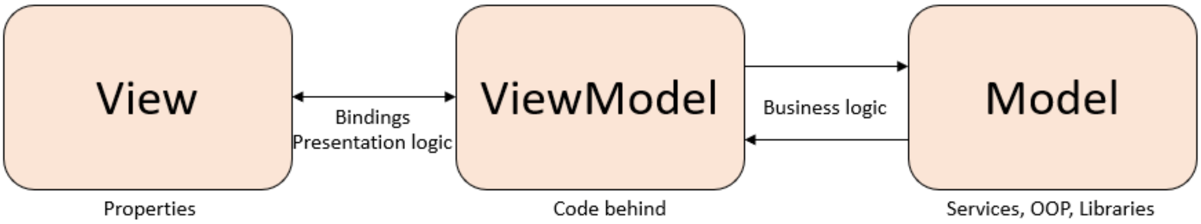


Figure 55 MVVM principle

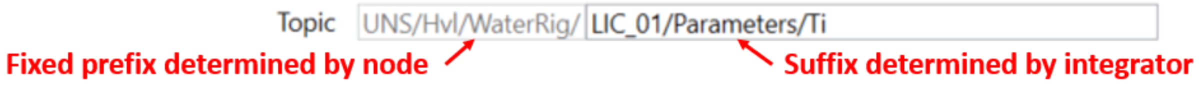
The third most important feature is the graphical user interface which is based on MVVM architecture. MVVM emphasizes decoupling between Views, which is what the user sees on the monitor, and the code behind it, which is implemented in classes named ViewModels. The View itself performs no logic. All Views have a corresponding ViewModel for organizing the code behind them. The advantage of handling all the logic in the code behind is that you have complete control and can achieve exactly what is needed. An example is the TreeView data structure described in detail in the appendix used to browse the OPC server address space. It uses lazy loading to reduce OPC server communication and handles node storage, hierarchy, and relationships. Logic not specific to a single View, for instance data access is implemented in classes called Models. Models can be placed in a library and reused in various applications. The advantage of using MVVM design principles is that the UI can be designed based on desired functionality, and the logic behind it is modularized for easy maintenance or modifications. All graphical controls are assigned names unique on the specific View and later bound to properties defined on the ViewModel.

4.2.5.2 Restrictions imposed by the gateway

The organization of the data and its presentation are the biggest challenges experienced by system integrators. Data published to the namespace are often unknown to the consumers before arrival from the UNS. Possibilities exist to create algorithms that search through messages for information. Still, the disadvantage is the computational cost lowering the upper limit of variables, often called tags, possible to pass through the network, resulting in reduced scalability. Therefore, much of the responsibility is on data producers. The gateway is designed with information modeling restrictions and standardized information attributes to minimize integration time. The same principles of a recognizable frame format as on protocol messages like IP packets are also enforced on gateway-published data.

4.2.5.3 Adding context

Context to the information is partially solved with a topic structure where the prefix is determined by the node which produced the information, e.g. **UNS/Hvl/Waterrig/**. The rest of the topic is up to the integrator to specify. One recommended strategy is that the hierarchy used to organize data on the PLC is re-used, e.g. UNS/Hvl/Waterrig/**LT_01/Status/Level** is used for a parameter belonging to level controller LIC_01. With such a structure, it is clear where the information originated. It can also be expanded to an infinite number of nodes without the risk of redesigning the topic structure because of naming collisions.

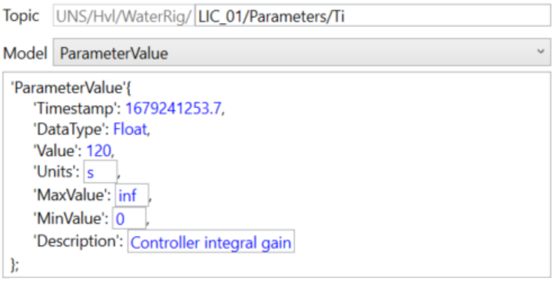

Topic UNS/Hvl/WaterRig/ LIC_01/Parameters/Ti

Fixed prefix determined by node

Suffix determined by integrator

Figure 56 Topic name is partially fixed and partially configured by the integrator

The other important part of the data context is the metadata transmitted with the changing values. Optimally, there would be a common agreement on which attributes should accompany data so that other systems can effectively utilize the information. The attached metadata simplifies external integration and can enable automatic scaling or parameter limitations. We have tried to exemplify how it can be solved by implementing fixed information models with attributes having clear meanings and fixed names. It is in line with the OPC Foundation, which has defined companion specifications [33] that, for example, define how robotics devices should package and transmit data. The companion models can easily be added to the template system on which the gateway is built. When configuring a new mapping, the user is forced to choose among one of the possible information models, partially determined by the node data type.



Topic UNS/Hvl/WaterRig/ LIC_01/Parameters/Ti

Model ParameterValue

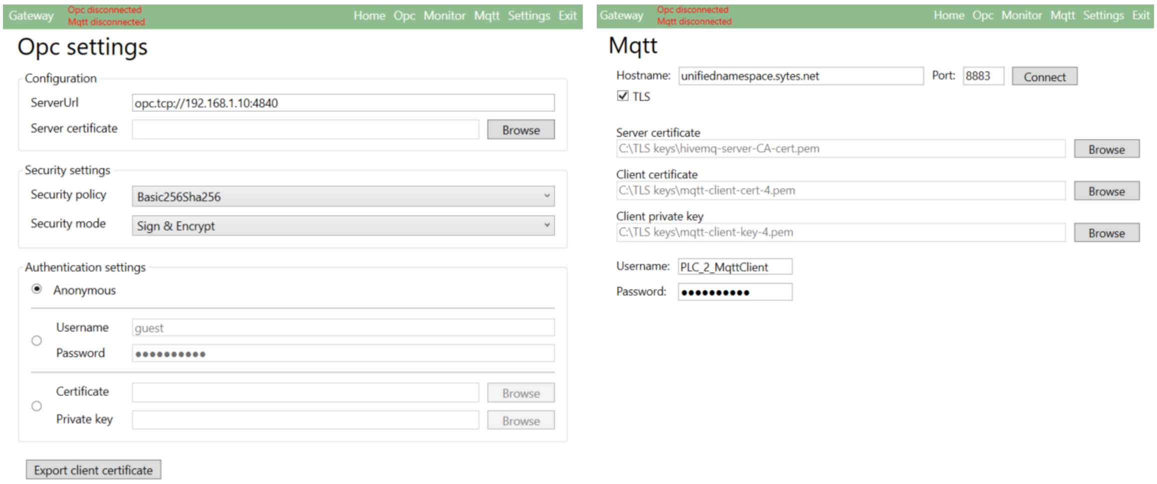
```
'ParameterValue'{
    'Timestamp': 1679241253.7,
    'DataType': Float,
    'Value': 120,
    'Units': s,
    'MaxValue': inf,
    'MinValue': 0,
    'Description': Controller integral gain
};
```

Figure 57 OPC Gateway restricted information models

4.2.5.4 Secure communication

It has become the norm that devices only offer encrypted and authenticated endpoints, even on local networks, which can be significant in size as the number of connected devices increases. The de facto standard in use today is X509 certificates. This invariable requirement of secure communication has also forced us to implement logic for handling certificates used for authentication and encryption. Well-tested libraries from Microsoft are used to simplify integration and to avoid beginner mistakes that could compromise the certificates and thereby the application security. To further improve security, the OPC library is designed to be easily integrated with the Windows certificate store. The advantage is that the storage of certificates and keys is protected against attacks executed through runtime memory on the host since the library is designed to prevent those kinds of leaks.

Alternatives to choose between different authentication credentials and types of certificates are included to enable the gateway to be connected to a wide range of device endpoints. The security against each endpoint is independent of the other. It ensures that if the OPC or MQTT certificates are compromised, only the connection to this endpoint will be affected.



Opc settings

Configuration
ServerUrl: opc.tcp://192.168.1.10:4840
Server certificate:

Security settings
Security policy: Basic256Sha256
Security mode: Sign & Encrypt

Authentication settings
● Anonymous
○ Username: guest
Password: ●●●●●●●●●●
○ Certificate:
Private key:

Mqtt

Hostname: unifiednamespace.sytes.net Port: 8883
☑ TLS

Server certificate: C:\TLS keys\hivemq-server-CA-cert.pem
Client certificate: C:\TLS keys\mqtt-client-cert-4.pem
Client private key: C:\TLS keys\mqtt-client-key-4.pem

Username: PLC_2_MqttClient
Password: ●●●●●●●●●●

Figure 58 OPC and MQTT security alternatives selected in gateway application

4.2.5.5 Robustness

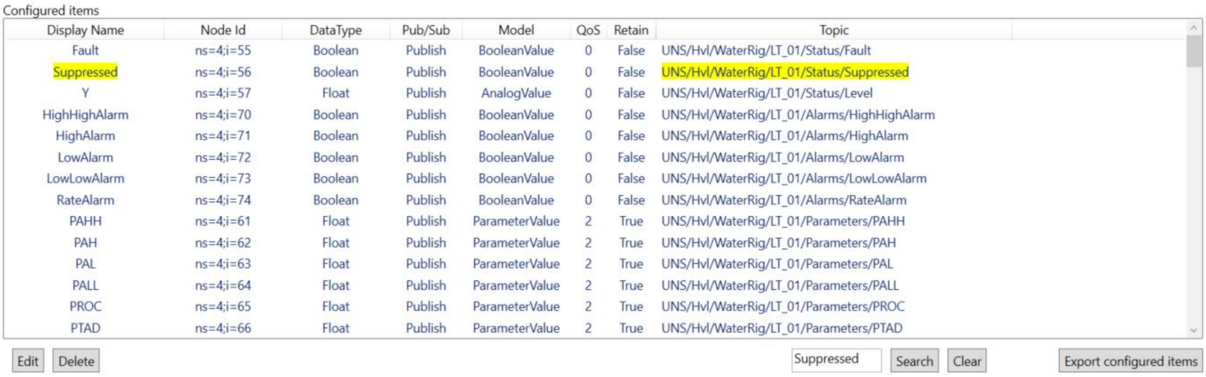
Unforeseen events can occur when a system is in operation. Power outages can cause the host device to restart, or the connections that the gateway maintains to the remote endpoints can be faulted due to network issues or server restarts. It is critical that the online state is restored after errors have disappeared without requiring actions from the system administrator.

The gateway is configured with the option to start at Windows startup. All configurations executed while online are continuously stored in the offline configuration file in addition to the runtime memory. Plan or unforeseen application or host reboot will read from this file and thus continue with the previous setup.

The connections to the endpoints are stateful and monitored to detect network issues. An event will fire to notify if the connection is malfunctioning and triggers a re-connect cycle that will continuously attempt to re-connect with a periodic interval. The previous session will be re-establish after a successful re-connection to either of the two endpoints with all the previous parameters still active.

4.2.5.6 Configuration Overview

It is possible to see a complete list of configured mappings by navigating to the Monitor view. The list can be searched based on the Display name, Node Id, or Topic name to identify already configured items. Existing configured items can be edited or deleted from the list of items using the buttons below. Finally, the complete list can be exported as a .csv file for easy distribution to external partners that will interface with the system.



Configured items

Display Name	Node Id	DataType	Pub/Sub	Model	QoS	Retain	Topic
Fault	ns=4;i=55	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Status/Fault
Suppressed	ns=4;i=56	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Status/Suppressed
Y	ns=4;i=57	Float	Publish	AnalogValue	0	False	UNS/Hvl/WaterRig/LT_01/Status/Level
HighHighAlarm	ns=4;i=70	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Alarms/HighHighAlarm
HighAlarm	ns=4;i=71	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Alarms/HighAlarm
LowAlarm	ns=4;i=72	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Alarms/LowAlarm
LowLowAlarm	ns=4;i=73	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Alarms/LowLowAlarm
RateAlarm	ns=4;i=74	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Alarms/RateAlarm
PAHH	ns=4;i=61	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PAHH
PAH	ns=4;i=62	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PAH
PAL	ns=4;i=63	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PAL
PALL	ns=4;i=64	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PALL
PROC	ns=4;i=65	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PROC
PTAD	ns=4;i=66	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PTAD

Search field: Suppressed

Figure 59 Overview of currently configured mappings in gateway application

4.2.6 Ignition Gateway

After gaining experience in building a gateway from scratch, it was time to explore the recommended solution: to use already existing technology on the market. Not all products have drivers that support MQTT transmission protocol, and not all products are open to the public without expensive licensing costs. Ignition is a SCADA alternative developed by a robust team that continuously maintains the product and which has put effort into integrating MQTT via the SparkplugB specification. It has the option of a 2-hour free trial which can be restarted an unlimited number of times. The trial provides access to all of Ignition's functionality.

4.2.6.1 SparkplugB

MQTT was initially developed as the backbone for communication in SCADA systems deployed in unreliable or constrained network environments. One of the success criteria was the freedom to send anything anywhere and the possibility of device state management. The downside is the lack of standardization when different systems are to be connected, which has demanded extensive manual integrations. SparkplugB tries to overcome this shortcoming by limiting the freedom of topic names and payload content and by using the underlying functions of MQTT to keep track of the nodes and devices online state. Simply put, MQTT defines how the transport of packages should take place, and Sparkplug defines a set of rules for how information should be organized. SparkplugB adds no new functionality but limits the freedom that exists within MQTT so that devices that follow the specification can be more easily integrated. Not everyone follows the specification, but data can be exchanged much more efficiently for those who do.

4.2.6.2 Cirrus link modules

Cirrus Link is an organization focused on developing Ignition modules to add SparkplugB encoded messages over MQTT as a data source into Ignition. There are three different modules of interest.

The Transmission module is designed to publish variables, known as tags, organized into tag sets to a device topic in the UNS, e.g. **spBv1.0/UNS/…/PLC_2**. The source of the tags can be any local connection configured inside the Ignition gateway. Our project fetches data from the OPC connection, making it available to anyone with UNS access via the MQTT broker cluster.

The second module of interest is Engine, an MQTT subscriber, subscribing to the Transmitter published tag sets, making them available as data sources on the SCADA system's host. It also includes possibilities for publishing commands to any MQTT topic.

Distributor is a module that is an MQTT broker. Suppose we had not set up our own data broker. In that case, the module is an alternative made by the same manufacturer and thus integrates specifically to support the other Cirrus link SparkplugB modules.

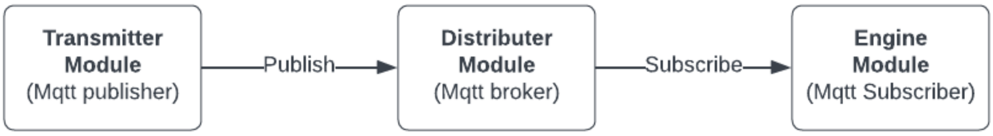


Figure 60 Interaction between Cirrus Link Modules

4.2.6.3 OPC connection

The connection to the OPC server endpoint is configured in the Ignition gateway, including the mutual import of certificates into Ignition and the PLC's OPC server. It is then possible to go online to choose which tags are to be added to the various tag sets that are distributed using the Transmitter module.

4.2.6.4 TAG sets

Tags (variables) are organized into tag sets. Information can be separated into different tag sets to enable different authorization levels for specific tag groups. Drag and drop during online browsing can add anything from a single tag to the complete device into published tag sets. The possibility of defining User Defined Datatypes (UDTs) and adding metadata to tags further improves the context before publishing to UNS. Ideally, all required context should be added before publishing to be faithful to the single source of truth from the edge principle.

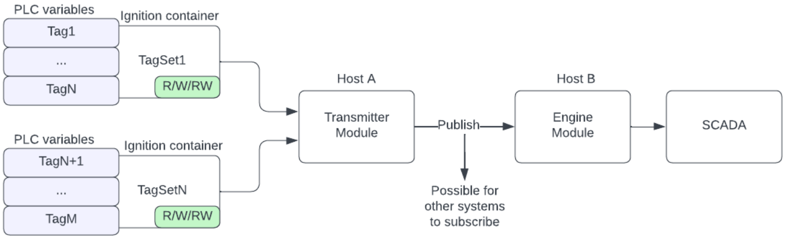


Figure 61 Tag organization, access control, and distribution of tag sets inside Ignition

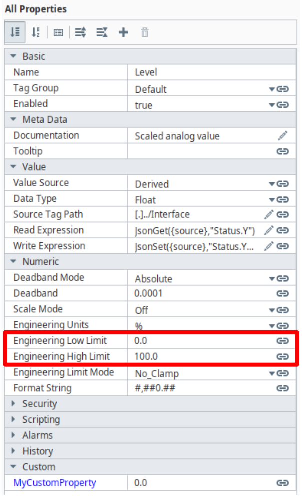
4.2.6.5 User Defined Datatypes

Ideally, entire data structures should be automatically mapped to objects when configuring new tag sets. Ignition is continuously developing support for partner devices. Unfortunately, Siemens devices are not supported at the time of writing this thesis. A workaround is to define the UDTs manually. Then any object matching the same attributes can be mapped to the blueprint, and a single reference is sufficient. It is not a major drawback since most objects in PLCs and SCADA systems often follow a predefined template to efficiently be reused in new projects resulting in a small number of object classes. These same templates are implemented as UDTs in the Ignition gateway.

An advantage of the SparkplugB specification is the option to distribute UDT templates as part of the birth message. Only the transmitter module is required to define UDTs. Based on the birth message, the UDTs will automatically be generated in an Engine module. This is an important step towards making information plug-and-play via online browsing.

4.2.6.6 Single source of truth

Often, variables are created to parameterize objects defined in edge devices. For example, the AnalogValue object has a parameter PYHR which is the upper limit that the instantiated value can take, enabling the same object to be utilized for different measurements having different ranges. There is value in sending such parameters as metadata associated with the analog value itself, enabling scaling or limiting values to be automatic in the system that utilizes them. Ignition designer has standardized properties when configuring new tags, which are assigned and included when the data is distributed. In addition, custom properties can be added as needed. A possibility is to add bindings that automatically update metadata when it changes in


All Properties

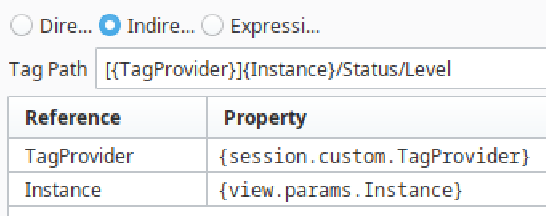
Basic	
Name	Level
Tag Group	Default
Enabled	true
Meta Data	
Documentation	Scaled analog value
Tooltip	
Value	
Value Source	Derived
Data Type	Float
Source Tag Path	[.]../Interface
Read Expression	JsonGet({source},"Status.Y")
Write Expression	JsonSet({source},"Status.Y...
Numeric	
Deadband Mode	Absolute
Deadband	0.0001
Scale Mode	Off
Engineering Units	%
Engineering Low Limit	0.0
Engineering High Limit	100.0
Engineering Limit Mode	No_Clamp
Format String	#,##0.##
Security	
Scripting	
Alarms	
History	
Custom	
MyCustomProperty	0.0

Figure 62 Tag metadata

the edge device where it originates. In our project, we have created this binding between PYHR/PYLR and the Engineering Low/High Limit for the Level variable.

4.2.6.7 Lowering integration time

It is common to define objects used on the HMI based on the same UDTs that store data in the PLC and are used by the Transmitter module when transferring objects. Defining these objects requires an initial effort, but then new objects of the same type can be easily added just by addressing the UDT root. All other references are derived from the root. Ignition simplifies the process even further by enabling drag-and-drop of tags directly from the Engine tag provider onto the Hmi canvas. The Tag provider is dynamically addressed inside objects to enable tags from different providers or to simplify the exchange of the default provider.



○ Dire... ● Indire... ○ Expressi...

Tag Path [{TagProvider}]{Instance}/Status/Level

Reference	Property
TagProvider	{session.custom.TagProvider}
Instance	{view.params.Instance}

Figure 63 Indirect Tag addressing in Ignition

4.2.6.8 Update frequencies

There is a compromise between rapidly updating changing variables and the use of bandwidth in a network. A SCADA system is traditionally intended to have an overall overview of the system with the possibility of sending commands or updated setpoints to the control devices that directly interface the physical process. It is not directly responsible for executing these commands on the physical hardware. Therefore, the real-time for SCADA is different than for a PLC, which can require sensors' and actuators' reaction times in the millisecond range. Consequently, there are options for how often a SCADA tag provider will update the values to balance the trade of.

Some of the parameters that determine update intervals are:

- PLCs are configured with minimum update interval to ensure the CPU has enough time to process between different requests. All changed monitored items will be updated if polled at this minimum interval.
- Clients subscribing to embedded PLC OPC servers are configured with a minimum poll interval.
- The transmitter module has the option to wait for a configured delay after the first tag is changed before all changed tags are transmitted in a combined SparkplugB message.
- Transmitting the entire PLC data set in one data block can be optimized since the time required to transmit more data can be less than transferring an increased number of messages. Reading the values of the message is efficiently done with pointers to fields inside this data block.
- Nested structure on Ignition pages is a penalty hit but is often required to achieve code reusability and maintainability.

4.2.6.9 Access control in Ignition

Restrictions should be placed on who has access to data published to the UNS. Unintended or malicious modification of tags could result in undesired or dangerous process states. Authorization is also configured inside the different Ignition modules to limit access only to authorized users. Tags can be separated into different tag sets to better control who has access to which tags. Each tag set has separate read, write, or read-write levels. The option to block all write operations in either the Transmitter, Engine, or Gateway can be an efficient method to prevent a client who should only be able to observe real-time status from writing new commands. In addition, read-write access can be configured on a tag-by-tag basis if there is a need for even finer-grained control.

4.2.6.10 Redundancy in Ignition

Component malfunction must be expected. A common method to guarantee the system's availability is having identical components in parallel, where a backup is ready to take over if the primary component fails. Two Ignition gateways can therefore be integrated into the same network with built-in functionality for fault monitoring and switchover. A prerequisite is that both gateways are configured identically. This has not been tried due to time limitations.

While redundancy on the local network is achieved with duplicated gateways, redundancy on the route between hosts is ensured with a robust network of brokers. One method is to connect to our transparent cluster of redundant brokers. Alternatively, Ignition has the option of configuring connections to several brokers. Data is only published to the preferred broker, but an alternative route to our free HiveMQ cloud broker has been configured if the cluster should go down. It is possible to publish simultaneously to both brokers, which may be relevant if certain clients do not have access to the main cluster because of access restrictions.

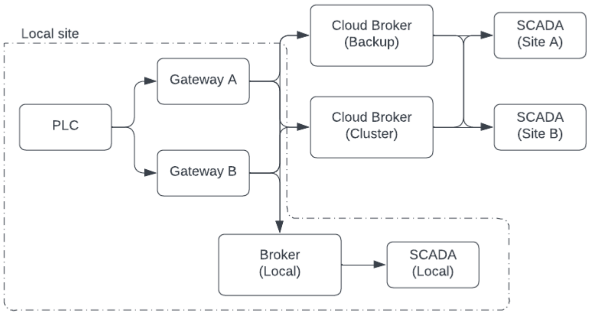


Figure 64 Ignition architecture for redundant systems

Several SCADA systems can be connected to the same architecture. It may be desirable for reasons such as the possibility to monitor and manipulate from different locations and not just for redundancy reasons. Among other things, it is common to have a SCADA placed locally to guarantee the possibility of controlling the system even if the link to the internet is down.

4.2.7 Design of UNS IIoT MQTT client

As a proof of concept, we wished to produce a hardware client to use in our MQTT network. Prebuilds Single board computers (SBCs) combined with off-the-shelf sensors would be cheaper than building simple standalone sensor arrays. However, we wanted to use the opportunity to explore PCB manufacturing in this thesis. During the time of the thesis, we manufactured 2 PCBs, one prototype and a final design. The final design successfully achieved connection to the UNS network with TLS and published its sensor update to the MQTT network. The design, decision-making and manufacture of these PCBs are thoroughly described in Appendix C.

4.2.7.1 Circuit design criteria's

The criteria set for our PCB was a hardware platform that could deploy sensors over the communication protocol I2C. The I2C protocol is intended to allow multiple digital ICs to communicate with a controller over two lines. The client must also be able to include the TLS handshakes and be a plug and play unit in LAN networks inside UNS. In order to be as versatile as possible, multiple power connections were added, with the added ability of power over ethernet. The PCB should be able to utilize ethernet and Wifi as potential communication bearers for the MQTT client. In order to control these various systems, a microcontroller (MCU) unit is required. This MCU should be easily programmed, and therefore a USB programmer should be added to the design to interface with the onboard microcontroller to enable the uploading of firmware without additional external programming hardware.

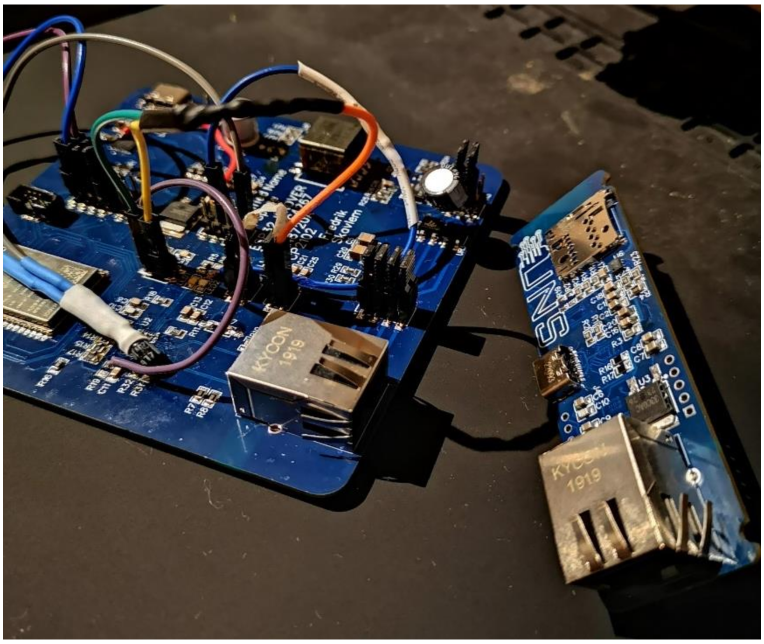


Figure 65 IIoT prototype left. IIoT final design right.

After the criteria's for the PCB were set, we started the PCB design process. The circuit design is primarily built from compiling the datasheets of the different modules on the PCB with some added inspiration from reference designs: [34-36] Further details with regards to component choices, reasoning for deviations from datasheets, solder techniques, layer-stackup, design philosophy and testing of the prototype can be found in appendix C.

One of the greatest challenges after conducting datasheet research was manually tracing each PCB. This proved to be a valuable experience, providing deep insights into the intricate process of PCB assembly. The PCB consisted of four layers, with the two inner layers serving as ground (GND) and interconnected by vias (connectors) to minimize potential electromagnetic interference (EMI), as explained in greater detail in Appendix C.

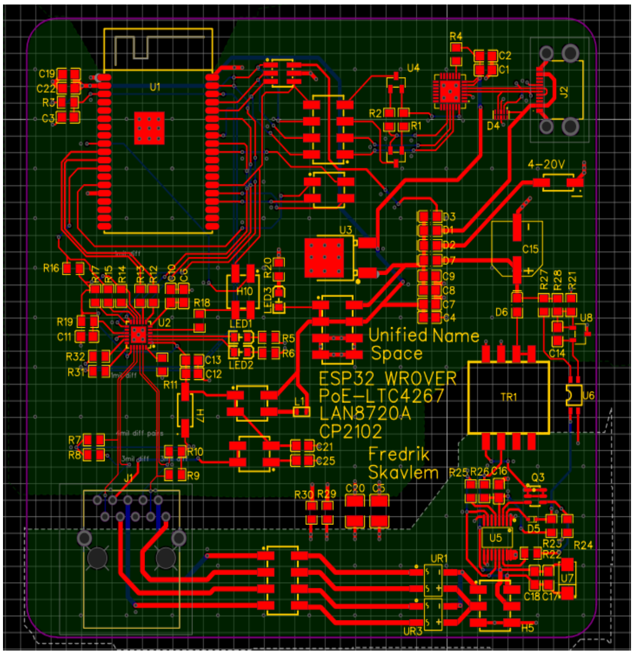


Figure 66 IIoT prototype trace, layer 1 red, layer 2-3 green, layer 4 blue

The final design was considered successful, as it was able to establish a connection to the MQTT cluster with TLS, despite a minor issue with the ethernet IC that caused a malfunction in the ethernet connector. However, the WIFI connection remained functional. We suspect that the malfunction occurred due to our manual removal of the LAN IC from the prototype PCB. The repeated heating of the IC to temperatures exceeding 300 degrees Celsius may have caused an internal failure within the LAN IC. Given additional time for investigation, we would have replaced the LAN IC with a new one to determine if it could resolve the issue.

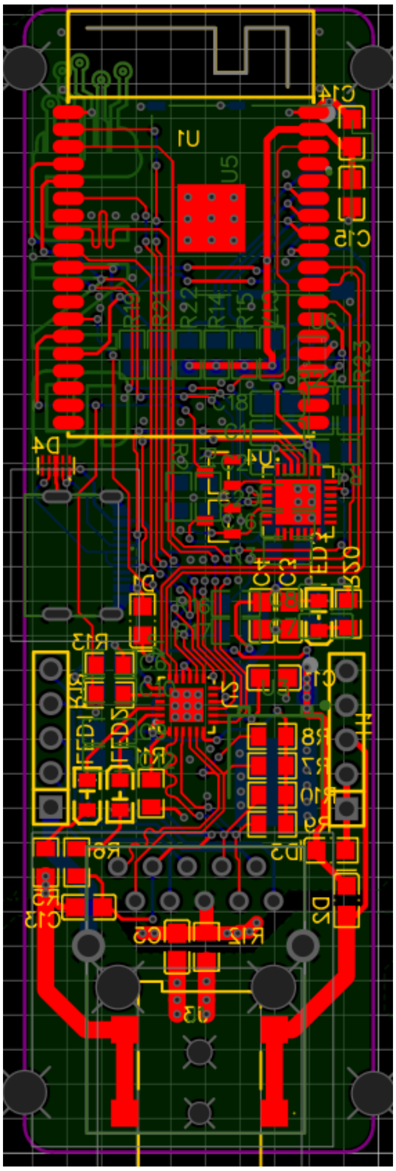


Figure 67 IIoT final design Trace, Red layer1, Green layer 2-3, Blue layer 4

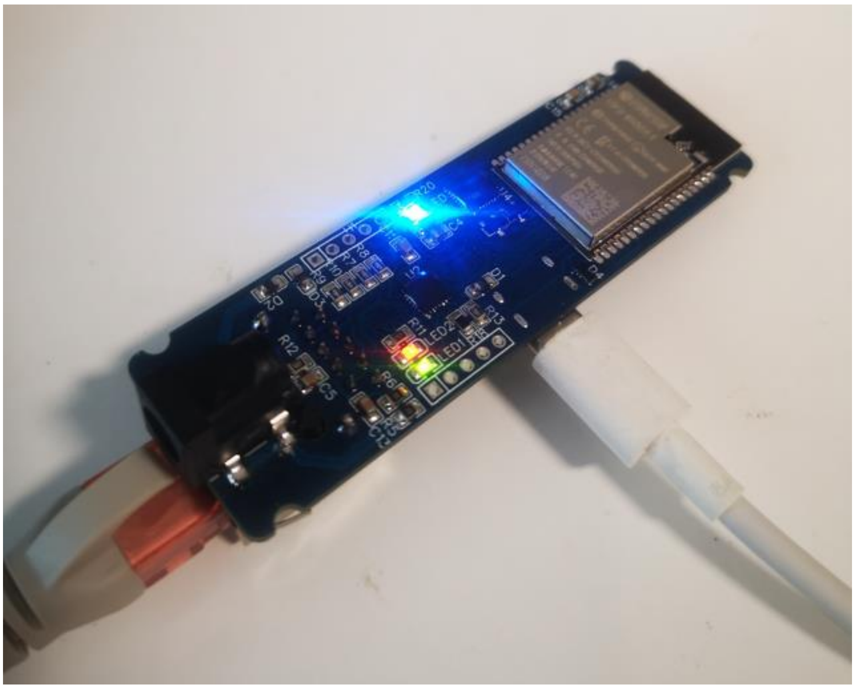


Figure 68 Testing of final IIoT design

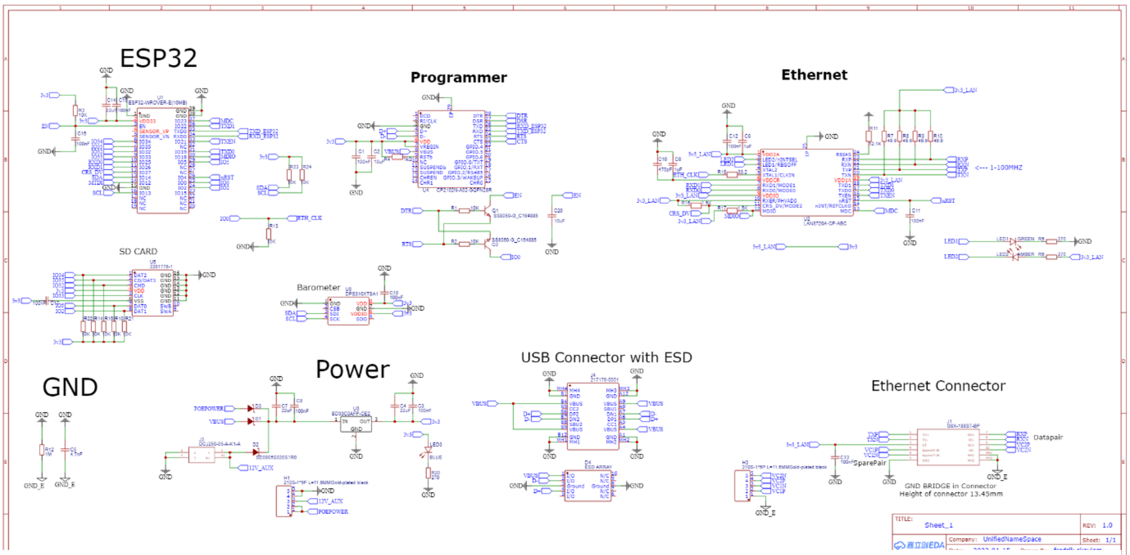


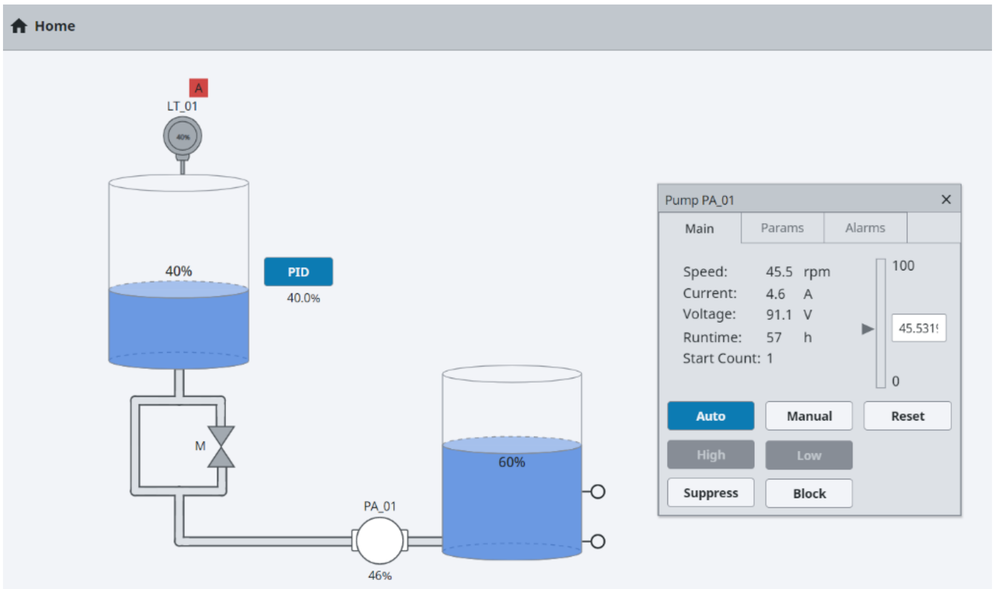
Figure 69 IIoT final design circuit schematic

4.3 Data consumers

Data consumers are devices that utilize data from the namespace to relay real-time state information to operators or perform tasks such as system performance analysis based on historical data collected during runtime. A prerequisite for consumer systems is gaining access to the information published by the data producers. Examples of data consumers include SCADA systems for monitoring and control, databases for long-term storage, dashboards for remote monitoring without system control capabilities, and analytic tools such as ML and AI. In this thesis, we have chosen to use Ignition software as our SCADA system. This chapter will start by describing how we have used Ignition as a SCADA host.

4.3.1 HVL Water rig SCADA

To exemplify data sharing over UNS, we have created a SCADA Hmi that subscribes to data published over MQTT from PLC_2, which controls the water rig simulator at HVL. The same PLC objects for an AnalogValue, Motor, Valve and PID controller have been defined in Ignition. New instances of these objects are added by dragging a new tag onto the page in Ignition Designer. The Ignition gateway resides on the same local network as PLC_2, publishing the tag set to the UNS. The SCADA host only requires an internet connection to the public address of the MQTT broker and can be situated anywhere. Real-time process status is displayed, and there is the option to send remote commands to the PLC. This setup has also been used for performance testing in chapter 5.



Pump PA_01

Speed: 45.5 rpm

Current: 4.6 A

Voltage: 91.1 V

Runtime: 57 h

Start Count: 1

Figure 70 HMI representation of HVL Water rig via Ignition

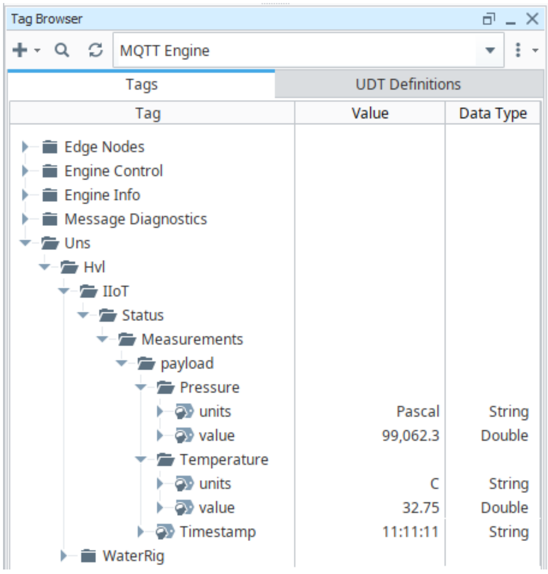
4.3.2 Object template definitions

Objects in the PLC are defined as classes, and this principle can also be implemented in SCADA systems to significantly reduce implementation costs. Therefore, all the functions defined in the PLC objects are also available on the Ignition template classes. All internal addressing within the object can be resolved by knowing only the root address of the instance. This simplifies the addition of new objects and reduces the risk of incorrect variable addressing, as only the root address needs to be configured when the object is instantiated. Only the most essential information is necessary and should be viewed on the Hmi pages as default. Minimal object symbols are the standard, with the option to open a pop-up window having the entire set of commands, statuses and parameters available.

4.3.3 Integrating IIoT devices

Integrating tag providers into SCADA systems built for the industry is often time-consuming and complicated. A thesis goal has been creating a system where arbitrary devices could easily communicate data via the UNS architecture, which other consumers could use. We will try to disprove this by integrating data from our own IoT device to display it on the Hmi and by sending commands in the opposite direction. All this will be done with the system in operation without any downtime.

The first step was to make tags from the IIoT device available in Ignition via the Engine module. By default, the Engine module is set up to receive messages only from the predefined SparkplugB namespace. However, expanding it with custom namespaces is easy even while the Gateway and Engine are in operation. As a result, the tag provider is grown with the most up-to-date values received from recently added UNS topics. The screenshot on the right shows the IIoT device added during online operation.



Tag Browser

MQTT Engine

Tags | UDT Definitions

```
Tag                                   Value      Data Type
▸ Edge Nodes
▸ Engine Control
▸ Engine Info
▸ Message Diagnostics
▾ Uns
  ▾ Hvl
    ▾ IIoT
      ▾ Status
        ▾ Measurements
          ▾ payload
            ▾ Pressure
              ▸ units                 Pascal     String
              ▸ value               99,062.3     Double
            ▾ Temperature
              ▸ units                      C     String
              ▸ value                  32.75     Double
            ▸ Timestamp             11:11:11     String
    ▸ WaterRig
```

Figure 72 IIoT device available as a data sources in Ignition

With the updated tag provider, previewing the format and structure of tags before implementing them in the SCADA system is elementary. Configuration can be done through a simple drag-and-drop interface, and additional metadata can be transferred with the tags to further enhance the context on the Hmi. As a result, integration times have been minimized. Values and units are automatically updated. Sending commands to the device for changing the units between Celsius and Fahrenheit has also been implemented as a proof of concept for reverse communication.

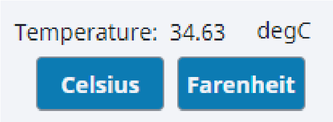
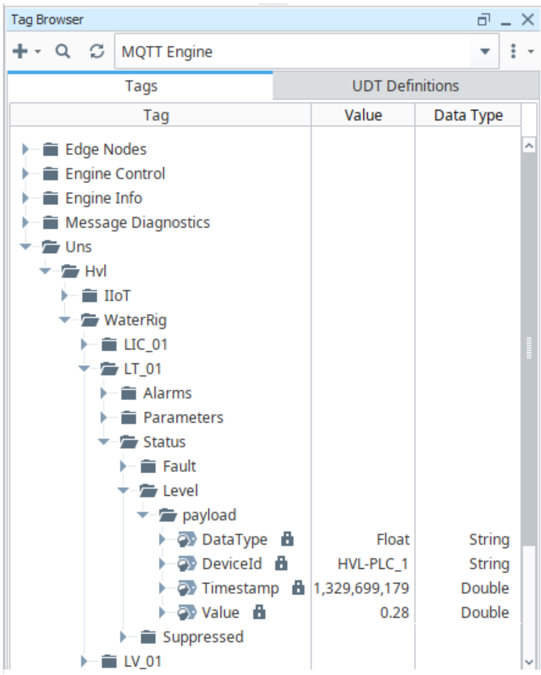


Figure 71 IIoT signal displayed on HMI

4.3.4 Integrating OpcUaGateway into Ignition

The self-developed gateway was created as a means to learn about the technology and gain experience with its opportunities and limitations. However, it was not used in the final system, where the intention was to demonstrate an off-the-shelf solution developed and maintained by others. Ignition, a fully developed gateway solution for sharing tags between a source and a SCADA system, was used to accomplish this. Nevertheless, this does not imply that the self-developed gateway does not work. In fact, we have successfully added it as a tag provider received by the Engine module located on the same host as the SCADA system. The tags are available in an organized structure, along with the additional metadata provided from the edge, which is part of simplifying integration.



Tag Browser

MQTT Engine

Tags | UDT Definitions

```
Tag                                   Value          Data Type
▸ Edge Nodes
▸ Engine Control
▸ Engine Info
▸ Message Diagnostics
▾ Uns
  ▾ Hvl
    ▸ IIoT
    ▾ WaterRig
      ▸ LIC_01
      ▾ LT_01
        ▸ Alarms
        ▸ Parameters
        ▾ Status
          ▸ Fault
          ▾ Level
            ▾ payload
              ▸ DataType              Float          String
              ▸ DeviceId              HVL-PLC_1      String
              ▸ Timestamp             1,329,699,179  Double
              ▸ Value                 0.28           Double
          ▸ Suppressed
      ▸ LV_01
```

Figure 73 PLC tags shared by OPC Gateway

4.4 Historian and data processing

In this coming chapter, we will delve into a critical aspect of our MQTT-based communication system: the Historian. Information passing through the broker is volatile, never being stored after delivery to the intended recipients. To address this, we developed a custom Historian from the ground up. Its primary function would be to handle the high volume of messages our HiveMQ cluster receives, filtering and storing relevant messages and topics into our database. The Historian is designed to retain new data and topics while avoiding storing redundant or unnecessary information. At the same time, it will maintain the data context of the received messages so that the information may be used for varying purposes later on. Throughout this chapter, we will explore the development and features of our Historian and discuss its significance in the larger context of our UNS broker cluster.

4.4.1 Preliminary

There are several existing Historian solutions available from various vendors. Still, creating our own Historian provides valuable insights into the technology and offers great control over the format and structure of our historical data. The first step in the process of creating our own Historian was identifying the dependencies and tools required to make everything work together. There are multiple ways of configuring dependencies and libraries in a Java project. The two we were familiar with was Maven, a mature, convention-based build tool with a vast plugin ecosystem, which is slower to compile and does not have the same flexibility as its counterpart. Gradle is a more modern, flexible, high-performance tool offering greater customization. Gradle also uses incremental builds and caching mechanisms to improve build performance. However, for our purposes, the configuration of each dependency was unnecessary, and the Historian itself was small enough not to worry about compilation times.

Maven offers a public pool of available repositories to check for new dependencies, including many versions, names and repository IDs. Fortunately, there are tools and websites available to simplify the process. One such website used extensively during the development of the Historian was the Sonatype Central Repository [37], an online storage and distribution platform for Java libraries, artifacts and dependencies. It is one of the most popular and widely used Maven repositories. In short, it allows developers easy access and incorporation of open-source libraries and artifacts, reducing errors and streamlining the build and deployment process.

While HiveMQ offers a Java library for MQTT communication and translation, which at the time seemed ideal due to our use of their product for our broker cluster, we quickly encountered issues with more advanced use-cases of the library, such as security implementation and use of TLS. Configuring and using the HiveMQ library was not as intuitive as we would have liked and simply took too much time. The documentation and code examples found online were hard to understand and lacked the details we desired. This resulted in us starting the search for other alternatives, with the most promising being Eclipse's Paho.

Given the challenges of the HiveMQ MQTT library, Paho was surprisingly easy to set up and start working with, even while using TLS encryption. Implementing Paho into our Java Historian took a similar amount of time as implementing HiveMQ-MQTT. However, Eclipse Paho had superior documentation, and being more widespread, finding useful code examples to draw inspiration from was less troublesome.

4.4.2 SSL and security within the Historian

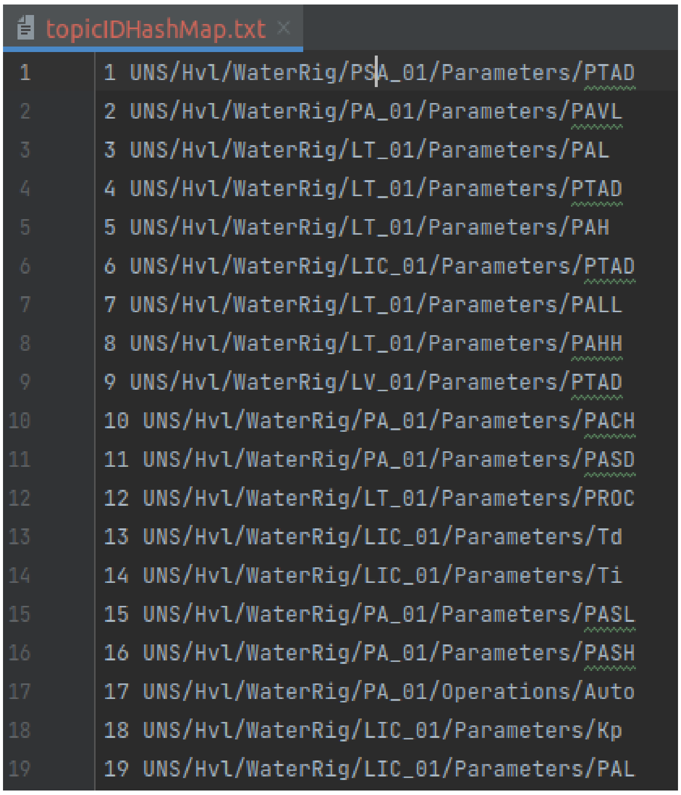
Ensuring the security of data passing through our Historian was a key priority in the design process. In line with our initial plans for generic code, we decided to create our own private certificates and keys. While this approach provided greater control and flexibility, it also required a significant amount of research about keys, certificates and general encryption that follows.

During the implementation of TLS in our HiveMQ cluster, we initially only generated PEM-keys for the two-way handshakes between our cluster and external applications. However, we encountered that Java was incompatible with these types of keys. Instead, we opt to use their specific keystore, which is compatible with all Java services and devices. To address this issue, we used OpenSSL to connect to our server and generate JKS-keys which the Historian could use. With this, we avoided developing our custom code, making PEM-keys work in a Java environment. Once the JKS keys were generated, authenticating and connecting with the use of PAHO was straightforward. This ensured the confidentiality and integrity of all data received and transmitted by the Historian.

With the secure connection established and SSL/TLS fully implemented, the next step was enabling information to flow into our database.

4.4.3 Historian logic and message handling.

When a message arrives via the PAHO client, background checks are performed to ensure minimal resource waste. This means that every time a message is received, the Historian already has some predefined information about existing messages and topics, as well as any new ones. Initially, our code disregards the message entirely, focusing solely on the topic assigned to the message. Upon starting the Historian, a text file is created to maintain data persistence between restarts. This text file stores each topic string received from the MQTT messages and assigns them a unique ID. To optimize the efficiency of reading and writing to the text file, the Historian reads the file during the startup phase and converts it into a HashMap stored in the heap. This approach primarily aims to improve read and write times while the application is running since reading and writing to a text file would be far too slow. Iterating over a text file has linear O(n) speed, meaning that search time increases with the file size. However, searching through a HashMap has constant speed O(1), ensuring constant search times regardless of the HashMap's size. The trade-off for this solution is a longer startup time, as the text file must be loaded into the HashMap. Our primary concern is processing speed during runtime, so a lengthier startup is an acceptable compromise.

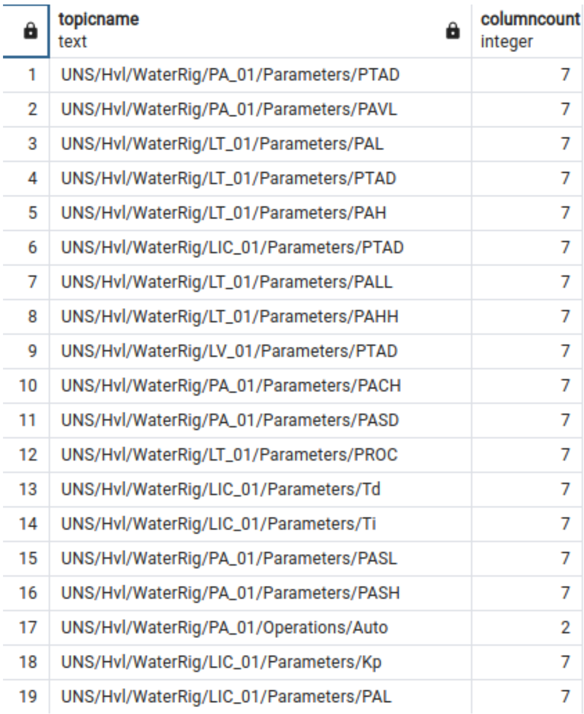


```
1 UNS/Hvl/WaterRig/PSA_01/Parameters/PTAD
2 UNS/Hvl/WaterRig/PA_01/Parameters/PAVL
3 UNS/Hvl/WaterRig/LT_01/Parameters/PAL
4 UNS/Hvl/WaterRig/LT_01/Parameters/PTAD
5 UNS/Hvl/WaterRig/LT_01/Parameters/PAH
6 UNS/Hvl/WaterRig/LIC_01/Parameters/PTAD
7 UNS/Hvl/WaterRig/LT_01/Parameters/PALL
8 UNS/Hvl/WaterRig/LT_01/Parameters/PAHH
9 UNS/Hvl/WaterRig/LV_01/Parameters/PTAD
10 UNS/Hvl/WaterRig/PA_01/Parameters/PACH
11 UNS/Hvl/WaterRig/PA_01/Parameters/PASD
12 UNS/Hvl/WaterRig/LT_01/Parameters/PROC
13 UNS/Hvl/WaterRig/LIC_01/Parameters/Td
14 UNS/Hvl/WaterRig/LIC_01/Parameters/Ti
15 UNS/Hvl/WaterRig/PA_01/Parameters/PASL
16 UNS/Hvl/WaterRig/PA_01/Parameters/PASH
17 UNS/Hvl/WaterRig/PA_01/Operations/Auto
18 UNS/Hvl/WaterRig/LIC_01/Parameters/Kp
19 UNS/Hvl/WaterRig/LIC_01/Parameters/PAL
```

Figure 74 Text file for storing historian topiclist on local host

After searching through the HashMap containing topics and their IDs, we determine whether the received topic is new. If it is, a new ID is generated for that topic and saved alongside it. Simultaneously, a unique schema in our database, "topiclist", which stores all topics and their respective IDs, is updated with the new topic and its ID.

First, we need a way to verify that no errors have occurred while storing these IDs; comparing our offline text file to our online database provides a quick insight into whether the Historian has been running correctly and if errors have occurred. Another reason is the schema and other identifier character limit of 63 set across different database management systems. Topic names could easily exceed this limit, so storing them in this manner allows for more flexible schema creation. A schema for storing topic names linked to IDs was our first logical choice and implementation. After updating the offline text file and online database with the unique ID and the corresponding topic, we can store information, or rather, our message payload, into the appropriate schema.



🔒	topicname text	🔒 columncount integer
1	UNS/Hvl/WaterRig/PA_01/Parameters/PTAD	7
2	UNS/Hvl/WaterRig/PA_01/Parameters/PAVL	7
3	UNS/Hvl/WaterRig/LT_01/Parameters/PAL	7
4	UNS/Hvl/WaterRig/LT_01/Parameters/PTAD	7
5	UNS/Hvl/WaterRig/LT_01/Parameters/PAH	7
6	UNS/Hvl/WaterRig/LIC_01/Parameters/PTAD	7
7	UNS/Hvl/WaterRig/LT_01/Parameters/PALL	7
8	UNS/Hvl/WaterRig/LT_01/Parameters/PAHH	7
9	UNS/Hvl/WaterRig/LV_01/Parameters/PTAD	7
10	UNS/Hvl/WaterRig/PA_01/Parameters/PACH	7
11	UNS/Hvl/WaterRig/PA_01/Parameters/PASD	7
12	UNS/Hvl/WaterRig/LT_01/Parameters/PROC	7
13	UNS/Hvl/WaterRig/LIC_01/Parameters/Td	7
14	UNS/Hvl/WaterRig/LIC_01/Parameters/Ti	7
15	UNS/Hvl/WaterRig/PA_01/Parameters/PASL	7
16	UNS/Hvl/WaterRig/PA_01/Parameters/PASH	7
17	UNS/Hvl/WaterRig/PA_01/Operations/Auto	2
18	UNS/Hvl/WaterRig/LIC_01/Parameters/Kp	7
19	UNS/Hvl/WaterRig/LIC_01/Parameters/PAL	7

Figure 75 List of topics, ID's and their column length in the database topiclist schema

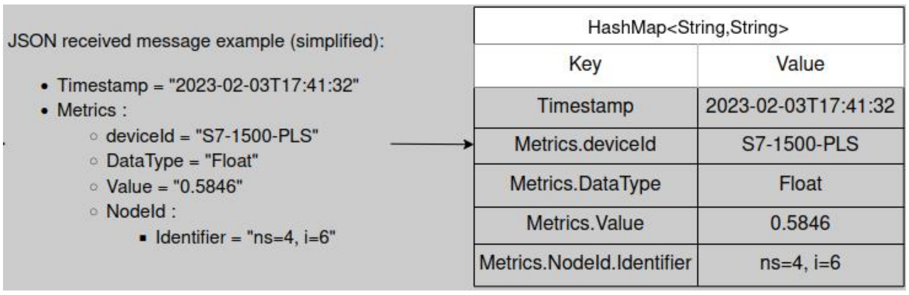
4.4.4 Deserialization of JSON

A question that arose after experimenting with various types of payloads sent by the clients and the publish and subscribe model was "What kind of structure do we need to enforce for the payloads?". Having everyone send data that was structured randomly would not be optimal for the historian, but it was necessary to provide users flexibility. PostgreSQL being a rational database, having every new client send differently structured data models would lead to the impossible task of managing the various data structures in unique tables. It would require a unified data structure to be agreed upon for all the various clients that would send and receive information in the unified namespace.

We decided that the best solution was to start with a generic format for sending information like JSON and work outwards towards supporting other formats when it was successfully implemented. Having agreed on this, next, we needed to determine how we could deserialize the incoming information as efficiently as possible.

Our initial idea was to encapsulate the incoming data within objects and then send it to our database in a manner customized for each distinct data type. However, this approach had several drawbacks. It was slow, inefficient, and required new methods for deserializing each incoming data structure which would differ from previous ones. The most obvious solution of boxing the JSON strings into objects and then unboxing them into a SQL query for every single message would be too time-consuming and inefficient in the long run. Instead, we agreed that the most efficient way would be to de-nest the JSON messages and to make query strings directly out of the payloads received.

We identified several libraries that could potentially solve our problems or, at the very least, provide tools to develop our own solution. The first library we tried was Jackson, by far the most popular JSON parser for Java. Although Jackson did not fully address our parsing issue, we were able to use part of its toolkit for our solution. The challenge we faced was that the information strings from the MQTT broker could have varying lengths. This meant that the code responsible for handling incoming JSON strings would need to support strings of any length. Another seemingly more complex issue was that the received JSON strings could contain lists. While this was not a major problem on its own, complexity increased as lists could be nested and comprised of more lists. The nesting could continue indefinitely and required de-nesting since our database could not store lists in any format. One workaround was to prefix every value within the list with the name of the list.



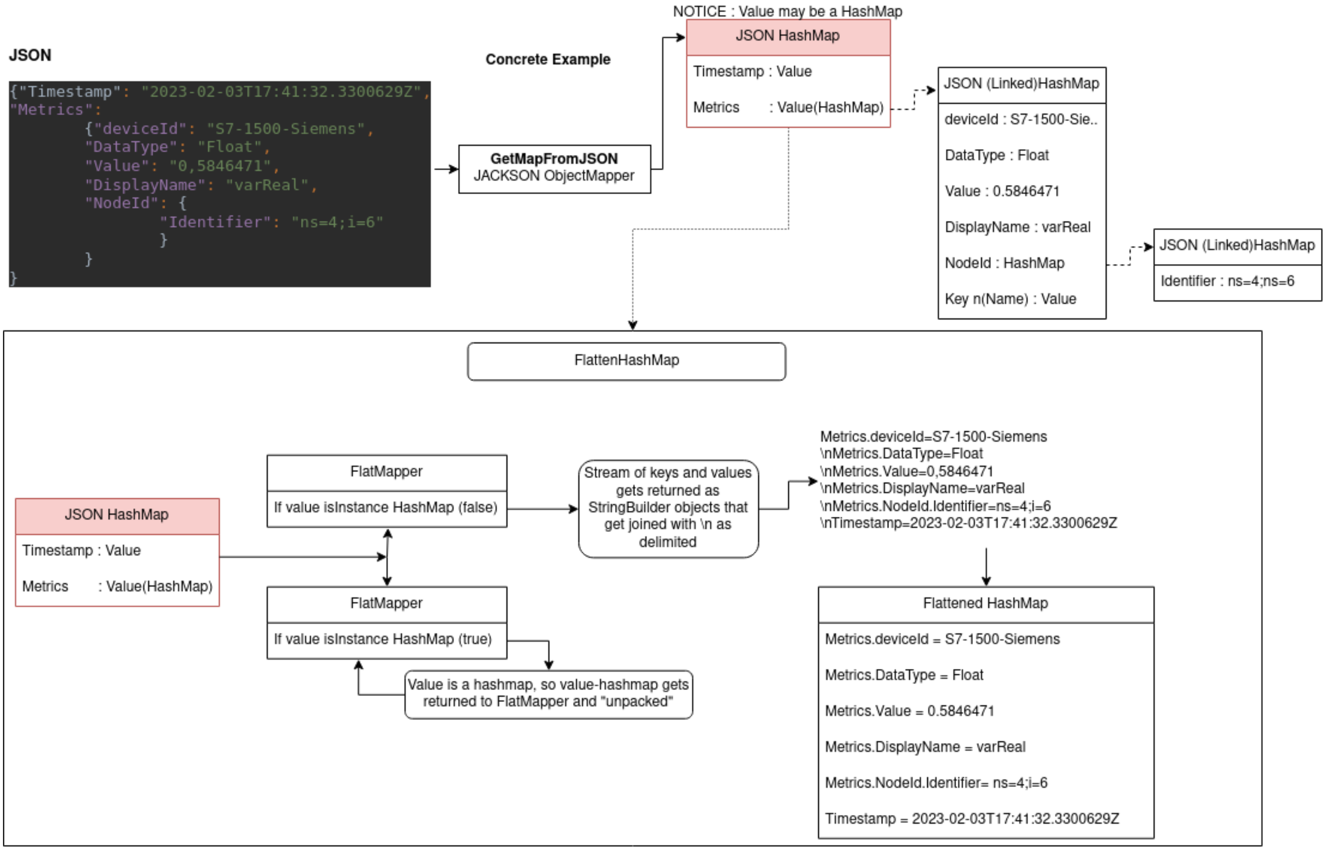
JSON received message example (simplified):

- Timestamp = "2023-02-03T17:41:32"
- Metrics :
  - deviceId = "S7-1500-PLS"
  - DataType = "Float"
  - Value = "0.5846"
  - NodeId :
    - Identifier = "ns=4, i=6"

HashMap<String,String>	
Key	Value
Timestamp	2023-02-03T17:41:32
Metrics.deviceId	S7-1500-PLS
Metrics.DataType	Float
Metrics.Value	0.5846
Metrics.NodeId.Identifier	ns=4, i=6

Figure 76 Flattening complex models by mapping attributes to HashMap keys in historian

Our solution involved writing recursive methods, which we will discuss briefly in the following paragraph and is explained in greater detail in the appendix. The FlattenHashMap and FlatMapper methods could be considered one but are separated to improve code readability. Starting from the beginning, where we receive a JSON string, we used Jackson's ObjectMapper library to convert raw JSON into a workable HashMap. ObjectMapper is optimized by employing streaming and buffering to minimize processing time. The GetMapFromJSON method takes the JSON-string as input, and initializes an ObjectMapper to handle the conversion from raw JSON data into a HashMap. At this point, we have HashMap with keys and values, where the latter is possibly a nested HashMap. To de-nest these, we use the FlatMapper method recursively, with it going through every value and entering into each HashMap it sees. When it is in the innermost HashMap, it starts de-nesting them on its way out toward the original HashMap. Once the original JSON-HashMap has been reconstructed and is no longer nested within itself, it is sent to the final method that transforms the HashMap into an SQL query ready to be dispatched to our database.

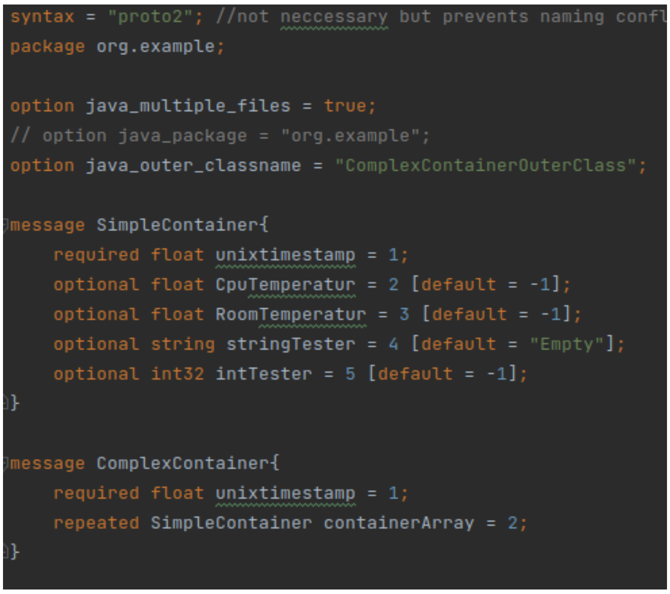

JSON
Concrete Example
NOTICE : Value may be a HashMap
JSON HashMap
Timestamp : Value
Metrics : Value(HashMap)
GetMapFromJSON
JACKSON ObjectMapper
JSON (Linked)HashMap
deviceId : S7-1500-Sie..
DataType : Float
Value : 0.5846471
DisplayName : varReal
NodeId : HashMap
Key n(Name) : Value
JSON (Linked)HashMap
Identifier : ns=4;ns=6
FlattenHashMap
JSON HashMap
Timestamp : Value
Metrics : Value(HashMap)
FlatMapper
If value isInstance HashMap (false)
FlatMapper
If value isInstance HashMap (true)
Value is a hashmap, so value-hashmap gets returned to FlatMapper and "unpacked"
Stream of keys and values gets returned as StringBuilder objects that get joined with \n as delimited
Metrics.deviceId=S7-1500-Siemens
\nMetrics.DataType=Float
\nMetrics.Value=0,5846471
\nMetrics.DisplayName=varReal
\nMetrics.NodeId.Identifier=ns=4;i=6
\nTimestamp=2023-02-03T17:41:32.3300629Z
Flattened HashMap
Metrics.deviceId = S7-1500-Siemens
Metrics.DataType = Float
Metrics.Value = 0.5846471
Metrics.DisplayName = varReal
Metrics.NodeId.Identifier= ns=4;i=6
Timestamp = 2023-02-03T17:41:32.3300629Z

```
{"Timestamp": "2023-02-03T17:41:32.3300629Z",
"Metrics":
        {"deviceId": "S7-1500-Siemens",
        "DataType": "Float",
        "Value": "0,5846471",
        "DisplayName": "varReal",
        "NodeId": {
                "Identifier": "ns=4;i=6"
                }
        }
}
```

Figure 77 Incoming message to historian flow diagram

4.4.5 Protobuf as encoding

We discovered Protobuf, a serialization/deserialization method showing great potential during our preliminary research for this thesis. By design, Protobuf efficiently transmits and serializes structured data between systems, making it a great match for our problem description. In short, this would mean that we only needed to define a structure for our messages once and that the publisher of the message controls the encoding and decoding format. As long as the clients receiving the messages can access the .proto file, it can automatically generate the code for encoding and decoding. This would make the system much more reliable and less error-prone, requiring less time to generate boilerplate code for encoding and decoding. While JSON still is a widely adopted and accessible format that offers more flexibility and can be easier to work with in some situations, Protobuf offers a more fluent way to implement the same message formatting over many different clients with varying purposes and architectures.



```
syntax = "proto2"; //not neccessary but prevents naming confl
package org.example;

option java_multiple_files = true;
// option java_package = "org.example";
option java_outer_classname = "ComplexContainerOuterClass";

message SimpleContainer{
    required float unixtimestamp = 1;
    optional float CpuTemperatur = 2 [default = -1];
    optional float RoomTemperatur = 3 [default = -1];
    optional string stringTester = 4 [default = "Empty"];
    optional int32 intTester = 5 [default = -1];
}

message ComplexContainer{
    required float unixtimestamp = 1;
    repeated SimpleContainer containerArray = 2;
}
```

Figure 78 Example .proto file

HiveMQ is designed to handle large volumes of real-time data streams, processing them quickly and efficiently. Many aspects of our project already prioritize performance, with Protobuf being a binary format that is more compact than text-based ones, like JSON. IIoT applications require structured data, high volumes and speed, making Protobuf a good solution. However, we were not interested in only using Protobuf exclusively, rather we wanted it as an addition on top of our already existing solution and implementation of JSON payloads.

Due to time constraints on this thesis, we only proved the viability of Protobuf implementation within our system and settled on only using JSON protocol for user-generated payloads.

4.4.6 Asynchronicity and R2DBC.

Since we wanted our Historian to run alongside the main cluster, it had to be able to perform two main actions asynchronously and independently of each other.

The first action was receiving a message from our MQTT cluster. Messages transmitted through the UNS could arrive at any time after our MQTT PAHO client had connected to our MQTT cluster. When a message does arrive, the PAHO client will jump to the “messageArrived” method. This method is run in the background on a different thread, making it a non-blocking method, allowing other code and methods to be executed simultaneously.

The second action we sought to execute independently and asynchronously involved storing the message in our database. After performing the checks mentioned in Section 4.4.3 and generating a fitting query for our database to execute, the next step was to implement queuing threads in a first-in, first-out (FIFO) order. One solution would be that when a query is complete, the lock class gets notified, allowing it to initiate the next query in the list. This method can be an effective way to achieve concurrency without an asynchronous SQL handler, which by default, Java does not provide. However, this solution’s scalability was uncertain. Consequently, a limited timeframe made us lean in favor of using R2DBC.

R2DBC, short for Reactive to Relational Database Connectivity, is a modern programming API designed for connecting and interacting with relational databases, in our case PostgreSQL, in a non-blocking manner. By using reactive programming principles, R2DBC enables applications to handle many concurrent database operations while maintaining efficient resource utilization. This approach is particularly beneficial in data-intensive applications, like our HiveMQ cluster, where traditional blocking JDBC connections may lead to performance bottlenecks.

R2DBC’s selling point is simple: it creates and manages its own thread pools, which can be more efficient in many cases. R2DBC establishes a pool of pre-initialized worker threads used to execute SQL queries concurrently. This approach reduces the overhead of thread creation and destruction, thereby optimizing system resource utilization by reusing threads instead of creating new ones for each query. Moreover, R2DBC is non-blocking, which enhances performance by enabling threads to handle multiple requests simultaneously. Within our project, R2DBC provides the functionality and speed needed to concurrently push messages into our database without blocking other parts of the Historian.

4.4.7 Information visualization

The primary goal and motivation behind integrating a Historian with our main HiveMQ cluster are to leverage the time-series and other valuable data processed through the cluster. We can gain valuable insights by storing, analyzing, and visualizing the information generated over time by the different devices. A centralized repository for this data can help us identify trends and patterns, both positive and negative, and serve as a preventive measure for detecting errors as early as possible. Utilizing a visualization tool in conjunction with our database enables us to visually represent the data stored in the historian in real-time, which can uncover trends, patterns, and relationships that might not be evident from the raw data.

Initially, we opted to visualize the data scraped by Prometheus using the open-source version of Grafana. This version still enabled us to query and display the desired information, mostly concerning the health of our cluster. Grafana presents various data through custom dashboards, which showcase real-time and historical data in different visual formats, such as graphs charts, tables and more. The panels that display the historical data can be customized further with various options for formatting, scaling and various filtering of the data. Grafana also supports data from numerous sources, including both time-series and relational databases. However, many of the supported databases and additional tools require extensions available only in Grafana's enterprise edition. As a result, while Grafana is a highly extensible and customizable platform, we were restricted to using only its base functionality.

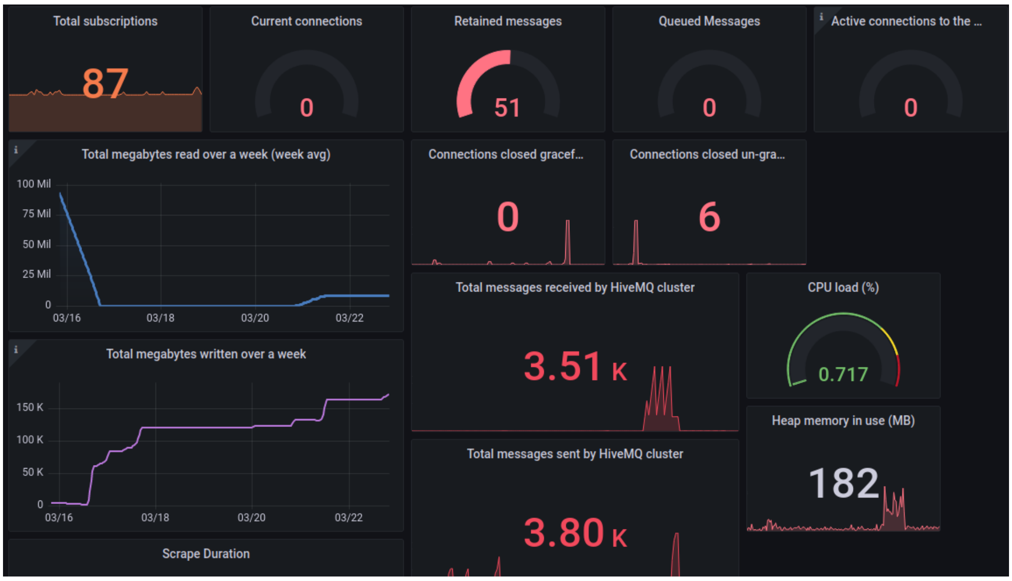

Total subscriptions
87
Current connections
0
Retained messages
51
Queued Messages
0
Active connections to the ...
0
Total megabytes read over a week (week avg)
100 Mil
75 Mil
50 Mil
25 Mil
0
03/16
03/18
03/20
03/22
Connections closed gracef...
0
Connections closed un-gra...
6
Total messages received by HiveMQ cluster
3.51 K
CPU load (%)
0.717
Total megabytes written over a week
150 K
100 K
50 K
0
03/16
03/18
03/20
03/22
Heap memory in use (MB)
182
Total messages sent by HiveMQ cluster
3.80 K
Scrape Duration

Figure 79 Grafana dashboard visualization of cluster statistics

Grafana is capable of visualizing the information stored in our database as well. Prior to which, we need to identify the appropriate table for any given topic stored in our topic list and sort the recorded values. Sorting can be by the time when stored in the database or the time it was recorded, both of which can identify potential issues in the system. The former being able to identify issues within the Historian and the latter within the connected devices responsible for recording the information in the first place. A prerequisite for efficient visualization is data standardization and sufficient data context. Making sense of data with much uncertainty and noise is hard and labor-intensive because of the required data transformations, which is particularly essential for third-party consumers. This will be covered in greater detail in 6.5, where we will discuss the use of machine learning to automate this contextualization of data.

5 Testing of the implemented architecture

Continuous uptime can be a decisive factor for the physical safety of an industrial automation system. Availability is often ranked as one of the most important factors for a technology to be accepted as a suitable solution in these situations. Cyber security is often ranked second to systems availability and is resolved by compromises such as isolation. It is also the reason why Goodtech desires these tests to be conducted. It is common to measure the performance of a system based on throughput, latency, robustness, and reliability, which are also natural groupings that we have used to divide the various tests.

5.1 Description of the test bed

The system architecture we have set up in our bachelor's thesis is designed to be easily scalable. We built it on the principle that it can be expanded in terms of the amount of hardware available, the capacity of the network connection from the ISP provider, and the number of nodes configured to handle simultaneous activity and provide redundancy. The idea is to pay for more resources if needed so that the system can continue to grow as demand increases.

Our system currently has very limited resources, particularly the broker solution running on Raspberry Pis. These devices barely have the resources needed for the broker application, so our setup must be considered a limited minimalistic setup rather than a realistic deployment. Therefore, tests such as throughput are not representative and should not be taken as definite limits of the system's capabilities. However, other tests such as reliability, detecting stale data, latency response times, and ensuring that the system does not get stuck in a state that requires manual intervention to restore normal operations, are still valid, even on our restricted test bed.

We conducted capacity and latency tests using a separately developed CUI client application that can send requests and log the results, including the number of requests, response time, and statistical information. We ran these tests using the client CUI application on a host outside of HVL to ensure traffic was on the Internet. The remaining hardware and reliability tests were on the LAN networks located at HVL.

Table 5 Hardware used during performance testing

PLC	SIEMENS 1516F-3 PN/DP
IGNITION GATEWAY HOST	Thinkpad T490 i5, 4 cores, 1.6Gz, 8GB RAM
FACTORY BROKER	Acer Aspire 5, 2.4GHz quadcore, 8GB RAM
CLUSTERED BROKER	RaspberryPi 4B 8GB RAM (Broker + Loadbalance) RaspberryPi 4B 4GB RAM (Broker) RaspberryPi 4B 2GB RAM (Broker) RaspberryPi 3B 1GB RAM x2 (out of resources)
HISTORIAN	Located on Factory Broker
ISP CONNECTION	20Mbit up/down

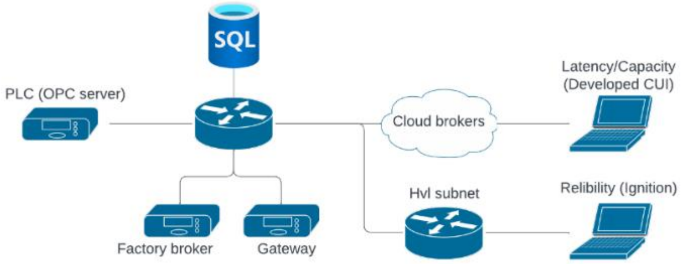


Figure 80 Configuration during system performance testing

5.2 Throughput, what is the system capacity

Throughput measures the traffic a system can transfer in a given unit of time. It will be both a measure of the capacity and will have consequences for the system's availability if all resources are consumed, resulting in clients having to wait before they can execute their requests.

5.2.1 Overloading the broker cluster

As the system approached completion at the end of April, we began conducting system-wide message overloading tests to assess the capacity of our system. Initially, we published as many message topics as possible to the factory broker to determine how they would transmit through the system via the bridge connection to the cluster. During this test, the cluster peaked at 24,500 messages per second. In the second test, we added multiple clients to the cluster and published as many messages on various topics as possible. The system peaked at 22,500 messages per second.

Test	Result	Comments
Cluster maximum incoming messages test1 **Method:** Use a single client to publish messages to the clustered brokers **Expected:** Record cluster maximum performance	24 500 msg/s	The ISP connection seems to be the limiting factor since the brokers still have available resources.
Cluster maximum incoming messages test2 **Method:** Use multiple clients to publish messages to the clustered brokers **Expected:** Record cluster maximum performance	22 500 msg/s	The ISP connection seems to be the limiting factor since the brokers still have available resources.

It's important to note that these results should be viewed cautiously, as our system was built on a student-funded budget. The cluster performs poorly due to being built on Raspberry Pi's. It was observed that the laptop with AMD64 architecture, 16GB RAM, and 2.4 GHz Quad core, hosting the factory broker outperformed our cluster. In addition, our school internet is subject to throttling and bandwidth limitations, and we never managed to achieve a higher inbound speed than 2 MB/s. Additionally, due to the restriction of having a trial license for HiveMQ, we cannot connect more than 25 clients at a time, so no mass client connection testing was possible.

In summary, we believe the main contributing factor to our poor test results is the internet connection since our system never reached a higher load than 32% during testing on one cluster node.

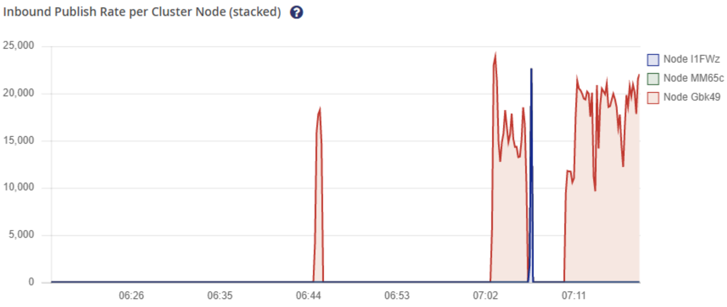
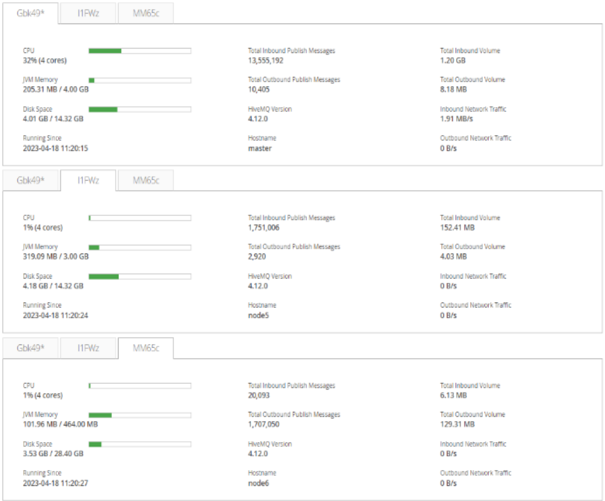


Figure 81 Inbound messages pr Second.
x axis local time, y axis messages pr second



CPU 32% (4 cores)
JVM Memory 205.31 MB / 4.00 GB
Disk Space 4.01 GB / 14.32 GB
Running Since 2023-04-18 11:20:15
Total Inbound Publish Messages 13,555,192
Total Outbound Publish Messages 10,405
HiveMQ Version 4.12.0
Hostname master
Total Inbound Volume 1.20 GB
Total Outbound Volume 8.18 MB
Inbound Network Traffic 1.91 MB/s
Outbound Network Traffic 0 B/s

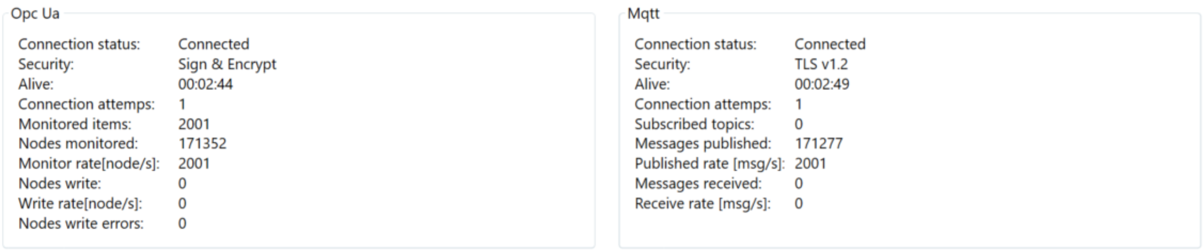
CPU 1% (4 cores)
JVM Memory 319.09 MB / 3.00 GB
Disk Space 4.18 GB / 14.32 GB
Running Since 2023-04-18 11:20:24
Total Inbound Publish Messages 1,751,006
Total Outbound Publish Messages 2,920
HiveMQ Version 4.12.0
Hostname node5
Total Inbound Volume 152.41 MB
Total Outbound Volume 4.03 MB
Inbound Network Traffic 0 B/s
Outbound Network Traffic 0 B/s

CPU 1% (4 cores)
JVM Memory 101.96 MB / 464.00 MB
Disk Space 3.53 GB / 28.40 GB
Running Since 2023-04-18 11:20:27
Total Inbound Publish Messages 20,093
Total Outbound Publish Messages 1,707,050
HiveMQ Version 4.12.0
Hostname node6
Total Inbound Volume 6.13 MB
Total Outbound Volume 129.31 MB
Inbound Network Traffic 0 B/s
Outbound Network Traffic 0 B/s

Figure 82 Cluster statistics during load testing

5.2.2 OPC UA server gateway loading

Tags are central in any project, making it simple to get statistics that can be used as a benchmark during the OPC server load testing. It is common for a project of significant size to have 2000 tags, based on a previous GoodTech project. It is also the recommended maximum for the 1516F CPU and is our target during testing. OPC server tags have been configured to change randomly on every PLC cycle scan and published on change. The minimum publish interval configured on the OPC server is 1000ms.



```
Opc Ua
Connection status:        Connected
Security:                 Sign & Encrypt
Alive:                    00:02:44
Connection attemps:       1
Monitored items:          2001
Nodes monitored:          171352
Monitor rate[node/s]:     2001
Nodes write:              0
Write rate[node/s]:       0
Nodes write errors:       0
```

```
Mqtt
Connection status:        Connected
Security:                 TLS v1.2
Alive:                    00:02:49
Connection attemps:       1
Subscribed topics:        0
Messages published:       171277
Published rate [msg/s]:   2001
Messages received:        0
Receive rate [msg/s]:     0
```

Figure 83 Connection statistics recorded by OPC gateway application

Test	Result	Comments
OPC Gateway monitored items test **Method:** Subscribe gateway to 2000/s updating items **Expected:** Record gateway performance	2000 item/s	Stable operation observed. We successfully transmitted messages in the reverse direction during this server load

Testing verifies that sending this amount of tags was not problematic, even under these somewhat worst-case circumstances. Commands were also sent in the opposite direction during the load testing to verify that the system was not overwhelmed. This amount of tags changing every cycle would be unlikely in a real system where most of the tags are static, like parameter values, boolean alarm bits, or states that do not change frequently. It must also be noted that all these tags have been handled as individual tags, each sent in its individual message. Most systems of this size use more efficient hardware with built-in communication processors than what we had available and are optimized to pack data into larger packets to utilize available capacity better. The key takeaway is that mapping between the OPC and MQTT interface has not been a bottleneck that would reduce the necessary performance in an actual deployment.

5.2.3 Maximum storage rate in the database

During load testing, the Historian exhibited no bugs or errors. However, a rational database like PostgreSQL is not designed to handle such a high load, as discussed in Chapter 6. The database quickly reached its default limit of 100 concurrent client connections when 2000 messages/s were tested.

Test	Result	Comments
Database storage test **Method:** Subscribe to 2000/s updating topics **Expected:** Record database performance	Failed	Messages are lost due to the database not being able to process them at the same rate as incoming messages

It does have to be mentioned that the Historian can fall behind in storing messages for a short time without any grave consequences. Suppose the Historian experiences traffic that is moving too fast for it to handle. In that case, it will start consuming memory to store these messages until they can be pushed into the database. There would likely be intervals during which the information flow would slow down, allowing the database to catch up.

5.3 Latency, how responsive is the system

Latency is a measure of the response time in a system. Different technologies are often categorized as being suitable or not according to whether they satisfy requirements for latency and determinism. It is therefore important to get some measures of what update times are expected for when commands are sent, until they are executed, and how long it takes before feedback is received. These tests must also be performed while the system throughput is being tested to verify response times in a system under a heavy load.

5.3.1 Host-to-host delay

There are several different sources of transport delay in a system, which all add to the total transport delay. It can be challenging to obtain accurate timestamps that indicate how long the transfer took since timing has to be acquired in different devices. To better understand the source of delays, it is therefore necessary to isolate individual components during testing to establish characteristics specific to them. One simple yet important test is the host-to-host delay, which measures the time it takes for a message to leave a host and arrive at the recipient, not including internal host processing. The test will measure time in flight and the time brokers use for routing messages.

The test can be conducted by having the same client publish and receive the messages, which makes timestamping more reliable. We have therefore created a console program that measures the time it takes for messages to be published and received. The test has been conducted while deliberately overwhelming the broker with a large volume of messages to simulate a realistic scenario. Various results have been observed during the trials. We have therefore accumulated results and calculated statistics such as min, max, average, variance, and skew.

The measured times also include delays occurring in routers and switches along the path between the client and the broker cluster. The test results indicate that messages are sent and received with minimal delay. These findings are not surprising, as storing messages is like a waiter collecting plates on their lap without handing them over. If the waiter does not remove the plates promptly, the stack will become overwhelming, eventually leading to a system breakdown. Similarly, brokers are intuitively expected to dispose of messages as quickly as possible to prevent accumulation.

Similar latency tests were conducted during cluster maximum load testing. The latency is significantly higher, but most are still within an acceptable range.

Test	Result	Comments
Host to Host latency test during medium cluster load **Method:** Publish a message from a client and subscribe to it as well. Record multiple tests to calculate statistics. **Expected:** Record latency time and statistics	Min: 3ms Max: 233ms Avg: 9.18ms Std: 5.516ms	Statistics calculated over 10.000 latency tested collected over 2 hours
Host to Host latency test during high cluster load **Method:** Publish a message from a client and subscribe to it as well. Record multiple tests to calculate statistics. Cluster is spammed as in load test 5.2.1 **Expected:** Record latency time and statistics	Min: 5ms Max: 5134ms Avg: 37ms Std: 50.51ms	Statistics calculated over 100 latency tests. Significant delays have been observed for a few messages. Overall most of the messages has an acceptable latency.

The majority of messages have a minimal delay. However, we do observe delays of several seconds for a small number of messages meaning that the results are skewed, caused by the minority outliers. This phenomenon is typical for non-deterministic networks and should be considered when assessing if the system is suitable for the task it is intended to solve. The results have been varying depending on the network connection. We have recorded average values of approximately 10ms latency and up to approximately ≈100ms on a network with poor bandwidth. The results in the report were during a day when the network activity was high at HVL.

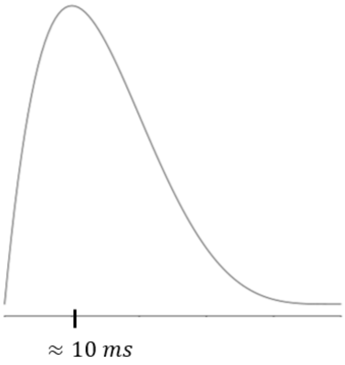


Figure 84 Example of a skewed distribution

Statistical properties are calculated over entire data sets. In practice, all results must be stored before measures such as average, variance, and skew can be calculated. It is often desirable to continuously update these variables as new data is produced to avoid memory overflow or to obtain continuously updated performance statistics. Incremental results can be used to detect failures or errors in the system, where a sudden increase in one of the statistical measures can be a symptom. We have implemented incremental algorithms partly for the learning part, even if we could have gotten away with accumulating results and then calculated statistics offline.

The disadvantage of online incremental updates is that rounding off small numbers can accumulate inaccuracies over time, as directly implementing the variance formula would have done. Welford's online algorithm [38] is designed to minimize rounding errors and is the one we have used to analyze our results. There is also an algorithm for incrementally updating skew, which is better than the naive method [38].

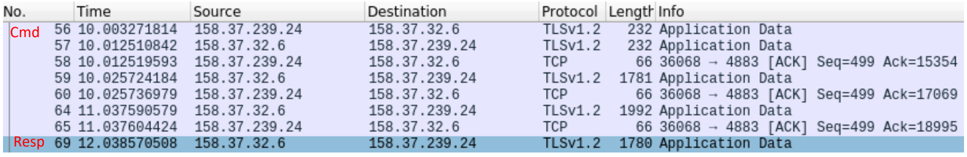
Sample variance: $s^2 = \frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n-1} = \frac{M_{2,n}}{n-1}$ where $M_{2,n} = M_{2,n-1} + (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n)$

Skew $= \frac{\sqrt{n}M_3}{M_2^{3/2}}$ where $M_2' = M_2 + (x-\bar{x})^2\frac{n-1}{n}$, $M_3' = M_3 + (x-\bar{x})^3\frac{(n-1)(n-2)}{n^2} - \frac{3(x-\bar{x})M_2}{n}$

5.3.2 Delayed processing between hosts

The total latency is the sum of all the processing and transport delays along the path from the transmitter to the receiver, including a round trip in the case of a command-to-feedback performance test. It's difficult to measure this time delay accurately. Still, we intend to document this using Wireshark (Network packet analyzer) to detect when packets leave the Ignition host and when the feedback packet returns. It is reasonable to assume that internal processing inside the ignition host is minimal, contributing negligibly to the overall latency. Encryption was disabled during this test.

Test	Result	Comments
Command-to-Feedback latency test **Method:** Execute a command. Use Wireshark to monitor the time until the host receives the feedback. **Expected:** Record latency time	2035ms	Online monitoring of the PLC estimates the command to be executed with minimal delay. Test result is from a single test.



```
No.       Time          Source          Destination     Protocol  Length Info
Cmd   56 10.003271814  158.37.239.24   158.37.32.6     TLSv1.2    232 Application Data
      57 10.012510842  158.37.32.6     158.37.239.24   TLSv1.2    232 Application Data
      58 10.012519593  158.37.239.24   158.37.32.6     TCP         66 36068 → 4883 [ACK] Seq=499 Ack=15354
      59 10.025724184  158.37.32.6     158.37.239.24   TLSv1.2   1781 Application Data
      60 10.025736979  158.37.239.24   158.37.32.6     TCP         66 36068 → 4883 [ACK] Seq=499 Ack=17069
      64 11.037590579  158.37.32.6     158.37.239.24   TLSv1.2   1992 Application Data
      65 11.037604424  158.37.239.24   158.37.32.6     TCP         66 36068 → 4883 [ACK] Seq=499 Ack=18995
Resp  69 12.038570508  158.37.32.6     158.37.239.24   TLSv1.2   1780 Application Data
```

Figure 85 Timestamps from wireshark during the command-to-feedback test

The test results reveal a delay of approximately two seconds between sending a command in TCP packet 56 and when the feedback was received in TCP packet 69. This is likely due to intentional delays introduced in the transmission path to increase the upper limit on the amount of information that can pass through the system. For instance, the PLC is configured with a maximum poll delay to ensure sufficient processing time between requests. Additionally, Ignition gateways are configured with a delay starting when the first tag is changed, transmitting all changed tags within this interval in a combined packet to save bandwidth. Theoretically, with our configuration and without taking TCP/IP best effort into account, the expected delay can be calculated to:

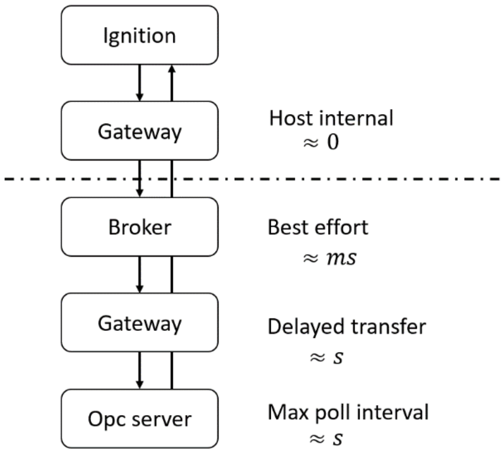

Ignition
Gateway
Host internal ≈ 0
Broker
Best effort $\approx ms$
Gateway
Delayed transfer $\approx s$
Opc server
Max poll interval $\approx s$

Figure 86 Different delays contribution to latency

Latency = OPC server poll rate + Transmitter pacing period = 1000ms + 500ms ≈ 1500ms

This theoretical result agrees with the actual measurements carried out on the H LAN. Identical tests have also been carried out on other networks with results that are slightly worse than these, especially the worst-case transmission time was significantly higher.

It is important to note that when monitoring the PLC in TIA Portal during command requests, minimal command delays were observed, suggesting that commands are transmitted almost instantly, and that TCP/IP best effort determines the transmission time. Thus, it is likely that most of the delay in the command-to-feedback test is incurred in the feedback part of the transmission.

5.4 Robustness, how will the system recover

Unforeseen events such as power outages or network failures must be anticipated. A system will have to re-establish all previous connections and continue to function as before after the sources of failure have disappeared without human intervention. The real-time status of the system is a critical component of the system's robustness to ensure device status is known and that data is handled as stale when devices are offline. Mechanisms to ensure that information or commands communicated during the downtime and essential for the system's continued operation are transmitted when the system is back online. Certain parts of a system are so critical that they cannot be allowed to fail. In such systems, it is common to install components in parallel, where one takes over if the other should fail.

5.4.1 Stale data

Tag quality and device online status should be marked on the HMI as is commonly done in existing SCADA systems that communicate over stateful connections to OPC servers. MQTT has no such connections between the producer and consumer of information. Stale data refers to data that is not updated due to a device being offline. Our testing involves verifying the presence of such mechanisms and tags in MQTT that are being updated to reflect the online state of the device, which can then be used to accurately display tag quality values on the HMI.

The test is fairly simple. It involves alternating between taking the OPC server and gateway offline and verifying in the engine tag provider and SCADA systems that the offline tags are properly marked.

5.4.1.1 OPC server offline test

The OPC session has parameters to monitor the connection quality. The most important of these being "Request Timeout". A client will set the connection as malfunctioning if no response to a request, such as a subscription query, is received within this timeframe. The same timeout is also used during the establishment of a session. The parameters must be set sufficiently to ensure that a connection attempt is not aborted too early, typically in the range of 10-60 seconds. After the OPC server was taken offline, we verified that this timeout governs when Ignition sets tags as stale, indicated by black frames and dots. In summary, this test reveals a delay of approximately 10 seconds (configured timeout) between when the OPC server was lost and before tags were marked appropriately on the HMI.

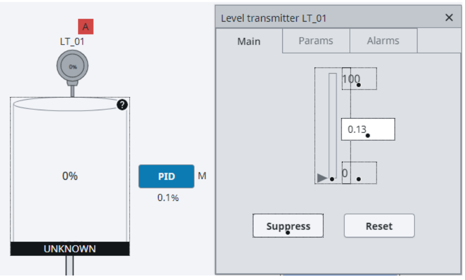


Figure 87 Ignition tags marked as uncertain when the OPC server is offline.

Test	Result	Comments
Stale data during OPC server offline period **Method:** Remove power from the OPC server **Expected:** Tags should be marked as bad quality on HMI	Passed	Tags are marked uncertain after configured OPC server timeout, set to 10 seconds in our system.

5.4.1.2 Gateway offline test

The time before tags are marked as faulty when the gateway is lost depends on how the disconnection occurs. If the gateway manages to disconnect gracefully, the tags are instantly marked as faulty and indicated with red squares and dots. However, if the connection is lost abruptly and the gateway did not manage to disconnect, then the Keep Alive time will govern when tags are marked as faulty. Currently, we have set the Keep Alive time to 5 seconds obtaining stable operations.

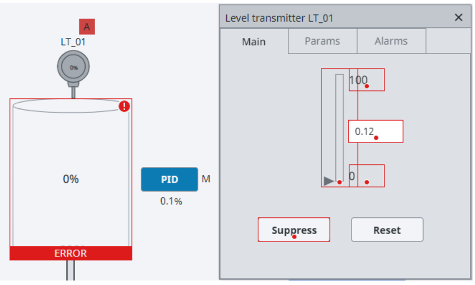


Figure 88 Ignition tags marked as faulty when the gateway is offline

Test	Result	Comments
Stale data during OPC server offline period **Method:** Remove power from OPC server **Expected:** Tags should be marked bad quality on HMI	Passed	Tags are marked faulty after MQTT broker Keep alive timeout, set to 5 seconds in our system.

State management tags are used to show the online status of nodes (gateways) devices (PLCs). During testing, we observed that the tag holding the gateway's online state changed and could be used on the HMI. However, the online state of the PLC device did not change at all. Based on this observation, we concluded that this is most likely a bug and not a limitation in the concept of device management.

To finalize the staleness test, we have verified that data is updated once the gateway, SCADA, or cluster is back online after an offline period. Tags were manipulated in the PLC during the offline period to ensure that the latest values were displayed on the SCADA system, even if the tags were no longer changing after the system was restored to the online state. Our tests have shown that this is achieved when using the Ignition modules, unlike our self-developed gateway, which requires a tag change or the PLC/gateway to be restarted when the SCADA system is back online. While a workaround to this issue is to publish tags as retain messages, we do not recommend it as a solution to prevent cluttering.

5.4.2 Redundancy

Redundancy tests are also straightforward to execute. They involve taking parallel devices offline and determining if the system is still operating while verifying any unexpected results occurred during the switch-over. In our system, it is mainly the brokers that are arranged in a redundant cluster. During testing, we were able to disconnect the individual worker nodes, and the connection held by the disconnected node was transferred to an available node seamlessly. Due to our limited hardware, our cluster only contains one load balancer. This creates a single point of failure if the load balancer is disconnected. In our testing, when the load balancer was taken offline, the connections held by an offline node did not transfer. Our recommendation to ensure high availability is to have a minimum of two load balancers for the cluster.

Test	Result	Comments
Loss of cluster node (not the load balancer) **Method:** Switch of power to the worker node currently holding the connection to the PLC OPC server **Expected:** Connection is transferred to an alternative worker node	Passed	The connection was transferred. Nothing was observed on the HMI during the changeover.
Loss of the load balancer **Method:** Switch of power to the load balancer **Expected:** System failure	Failed	No redundant load balancer in the system.

The possibility of registering several brokers or clusters in the gateway is not something that we have intended for the system architecture in our thesis but is an option in Ignition and will be included for completeness. HiveMQ is offering the possibility to launch a cloud broker for trial testing that we have registered as an alternative route for the Transmitter and Engine gateways. No information is communicated through the alternative path as long as the primary is operational. After taking down the HVL cluster, which is the primary broker, we verified that the connection between the OPC server and the ignition SCADA was still operational via the alternative route. On HMI, there was no marking of stale tags as during the offline test above. Still, we assume that there has been a short offline period in the time interval when the route was changed.

Test	Result	Comments
Loss factory and cluster broker **Method:** Switch of power to the factory broker/cluster **Expected:** Connection is transferred to the cloud broker	Passed	The connection was transferred. Nothing was observed on the HMI during the changeover.

PLC, Gateways, and SCADA hosts can be set up as redundant components but have not been accomplished because of hardware and time limitations during the thesis.

5.4.3 Automatic system recovery

Various errors can be the reason why a system stops working, like getting trapped in an infinite loop or operating with illegal parameters that makes the connection malfunction. These kinds of errors should be avoided but are not something we can thoroughly test in released software. What we will verify is that the systems automatically start up in an operational state after power outages. Only binary yes or no results will be recorded depending on whether the system restarted successfully. No other measures that could have been used to compare systems against each other are emphasized, since the offline period is the majority of the time interval. Comparative results are also determined mainly by the host startup time.

The test was to power cycle the devices one after the other. We verified that the system returned to a working state between each test. After carrying out the single tests, we switched off the power on all the devices and verified that they could be switched on again in any order.

Test	Result	Comments
OPC server automatic restart **Method:** Power cycle OPC server **Expected:** Tags are marked as stale in Ignition during server offline period. System automatically resumes normal operation when the server is back online.	Passed	Tags are set stale according to 5.3.1 OPC server restart is slow (≈70s) Finally, the entire system resumes the online state after server startup.
Ignition transmitter gateway automatic restart **Method:** Power cycle the host hosting the transmitter **Expected:** Tags are marked as stale in Ignition during gateway offline period. System automatically resumes normal operation when the gateway is back online.	Passed	Tags are set stale according to 5.3.1 Ignition gateway startup time is dependent on gateway host startup time. Finally, the entire system resumes the online state after gateway startup.
Ignition engine gateway automatic restart **Method:** Power cycle the host hosting the engine **Expected:** Tags are marked as stale in Ignition during gateway offline period. System automatically resumes normal operation when the gateway is back online.	Passed	Tags are set stale according to 5.3.1 Ignition gateway startup time is dependent on gateway host startup time. Finally, the entire system resumes the online state after server startup.
Cluster automatic restart **Method:** Power cycle the local broker and cluster **Expected:** Tags are marked as stale in Ignition during server offline period. System automatically resumes normal operation when brokers are back online.	Passed	After the complete disconnect of all cluster nodes and load-balancers. The system used approx. 1min to reestablish all connections and resume traffic after reconnection.

Test	Result	Comments
Verify arbitrary startup order **Method:** Power of OPC/Gateway. Startup in order: **1. OPC server** **2. Gateways** **Expected:** System successfully restarted	Passed	None
Verify arbitrary startup order **Method:** Power of OPC/Gateway. Startup in order: **1. Gateways** **2. OPC server** **Expected: System successfully restarted**	Passed	None
Verify cluster startup after edge devices **Method:** Power up broker when OPC/Gateway is online **Expected:** System successfully restarted	Passed	None

Our test concludes that startup can be performed in arbitrary order.

5.5 Reliability, can the system be trusted

The system's performance can be degraded by interference from internal or external parties. Malicious external intervention is partially resolved using encryption and authentication, preventing outsiders from manipulating the system. On the other hand, interference from actors on the inside can be unintentional transmittals of data to unintended and unexpected topics, thereby malfunctioning the system. Such incidents should be prevented by establishing access control that minimizes the possibility of these situations occurring.

5.5.1 Read-only access

It would be most optimal for edge devices to set authorization restrictions since then they can restrict tags that should not be manipulated externally without being dependent on a system administrator having performed as intended.

Test	Result	Comments
Read-only access inside edge device (PLC) **Method:** Attempt to write to a tag configured as read-only in the PLC **Expected:** Tag should not change. Error message presented	Passed	Error message given to the user on the HMI. It’s not possible to write to any tag, even if the tag is configured as write in the PLC if it is in the same structure as read-only tags.

Unfortunately, during testing we experienced problems when restricting individual tags in more complex data types from being written to. We believe the issue is that the entire complex structure is sent as one entity in a Sparkplug message and must have the same access level. A possible workaround would be to map variables individually or in smaller groups. It can be simplified to some extent by splitting complex data structures into smaller groups and reusing the templates. It should be considered since the system's integrity is so important.

Fortunately, sending the same tag provider on different topics is possible. Any of the Ignition modules can make the tag provider read-only, making it possible to create a read-only provider for systems that should not have the option of sending commands. Then, the SCADA system that sends commands and setpoints has its own topic that other devices cannot access. The disadvantage is that data are sent twice.

Encryption and authentication are considered secure technologies where most weaknesses are due to human errors. We assume that the system has been correctly configured when assessed as a viable solution and therefore do not see it as appropriate to carry out encryption tests. We do not intend to suggest that encryption or secure storage of certificates is unimportant. However, it can only be tested by personnel with in-depth knowledge of cybersecurity.

6 Discussion

6.1 Systems interoperability

The process of sharing information can be roughly divided into three levels: Physical, Transportation, and Representation, defined by the thesis authors. TCP/IP stack protocols have essentially standardized the physical and transport layers into the Ethernet and TCP/IP standards. Consequently, getting the information to the correct receiver is therefore trivial in most modern systems, as long as security or authorization does not prevent this.

Effective use of the information by the recipient is a much bigger problem. Since MQTT does not define a payload, anything can be packaged and transmitted through the broker cluster, making publishing information as simple as knowing the address of the recipient. Compared to the configuration of the transport parameters, most of the time in setting up a system today is spent on mapping inputs and outputs, generating data structures for storing data and creating interfaces towards other systems to exchange data. A big challenge in making data plug-and-play is that manufacturers with different requirements need a different set of models and attributes. Changing requirements demanding revised models is another obstacle making cross-vendor interoperability challenging.

Often, information is structured so that it is easy to visualize and interpret for humans making it easier for them to do manual system integration. On the other hand, the machine depends on a recognizable format for efficient and robust processing of incoming data. Most industrial systems usually require performance guarantees and often do not have any margin for an algorithm to search through data to predict the most optimal action. It will be necessary also to standardize encoding and structure for systems to interoperate more efficiently and safe.

SparkplugB attempts to resolve the interoperability issue by defining a generic Protobuf model that allows for nested structures enabling it to package and publish essentially any information model. The technic depends on the model blueprint being distributed to the recipients, which can be done using the SparkplugB birth message. It works well between the various modules made by Cirrus Link for Ignition, but one has to remember that these are made by the same suppliers who ensure both the sending and receiving of the information. Other manufacturers must familiarize themselves with Cirrus Link's philosophy of using SparkplugB when developing their own system, creating them to expect data in the same format. It must also be noted that although SparkplugB provides the opportunity to define a complex model's blueprint in the birth message, the user is still free to choose how the content inside the message is structured.

Definitions summarize the OPC UA standard, which is also a major disadvantage. Defining too many choices, including information models geared towards different industries, makes the standard complex and consequently reduces interoperability. The standard emphasizes developers' ability to define custom models. The disadvantage is the potential for an overwhelming number of models, which could easily end up in the thousands due to revisions and because of the fact that users can define any information structure they desire. One method of mitigating the downside of having many different models is that systems connected to the internet could automatically poll public repositories for the models' blueprint. Unconnected systems could depend on local servers containing public and company-specific models. One way or another, model blueprints should be distributed automatically to achieve efficiency interoperability.

A proposed solution compatible with some of the systems in production today, and between the two previously mentioned strategies could be to emphasize standardizing attribute names, functions, and object hierarchies. Data transmitted by the MQTT protocol are byte arrays, regardless of encoding or structure. After serialization and deserialization, it ends up as a long string of bits representing underlying variables encoded as Ints, Floats, ASCII characters, or any other defined datatype. Pointers to fixed positions inside this sequence or searching for key-value pairs are needed to extract the desired information. There is great flexibility in choosing what is included in a message when key-value pairs are used. The Ignition project, which has been part of this thesis, is an example where not all the attributes an object contains are utilized. Object in the Ignition Tag provider is received as document types containing a list of all the key-value pairs. Variables are extracted using Json parsers. Adding or removing attributes would not have affected the parsing. What is essential is that the path to the individual attribute is in a reproducible format, e.g., …**ObjectName.Parameters.PYHR** for a parameter (**P**) determining the output (**Y**) high (**H**) range (**R**). Changing the path would make it impossible to address before data has been received.

Resolving the address of a variable nested inside a potentially complex model can be seen as intuitive, at least to humans glossing over the Json structure. On the other hand, decoding a compressed message using aliases, fixed positions, and datatypes defined in model blueprints assumed to have been distributed to the recipient before message transfer can be quite more challenging without prior knowledge. Data encoding can be valuable, especially in memory-saving situations such as real numbers, which can be effectively compressed down to 8 bytes achieving a range of $\approx \pm 10^{308}$ and with a precision of 10^{-16} [39]. But it depends on a common agreement of serialization and formatting of data types. It must otherwise be balanced between size, making own definitions, and the simplicity of other systems to make sense of the information being transferred. Such encoding may have potential when used in connection with publicly agreed standards such as SparkplugB or when data is communicated between familiar devices, but it is another major obstacle that opposes a common platform for sharing information.

It has always been a guiding principle in the OPC UA standard not to specify the actual protocol implementations to make it more future-proof. MQTT seems to be increasingly used in the latest revisions of the specification. OPC traffic wrapped in MQTT data packets is still an independent language, meaning that not everyone can interface with this traffic. The IT system often has the most value in the data produced on the edge and commonly has better functionality for decoding different message formats than non OPC compatible automation systems, which may not have as much value from this data. OPC data in the cloud is definitely a step in the right direction, enabling IT systems to eavesdrop on the control signals.

The problem of being locked to certain manufacturers due to protocols not being compatible with each other has always been and still is the biggest challenge today. Although we have a well-functioning system using Ignition and SparkplugB, having shown that adding arbitrary topic sources like the IIoT device is simple, its effectiveness strongly depends on the Cirrus link modules, becoming somehow locked to the protocol. It is a significant drawback since few providers currently support the SparkplugB specification. It is worth mentioning that the system works very well and the way to get more people to adapt is perhaps to put it into use so that it gets better traction.

6.2 Implementation strategies

UNS intends to establish an architecture for sharing data in an enterprise with the idea that future devices will be connected requiring minimal integration time and without regard for the existing topology. This idea implies that the system should be planned and realized based on where the company wants to end up and not by considering existing networks and devices, which can result in the system being developed with compromises that will not work in the long run. Fortunately, the MQTT protocol is compatible with such a problem. It can easily be realized on existing TCP/IP and Ethernet networks. These networks are based on a client-server architecture where messages are addressed to specific devices. There is no single master controlling or dictating how communication flows. Different protocols can therefore coexist on the same network. Alternatively, a completely new network can be created. The advantage is the possibility of gradually building up the system without consideration of the existing system. The disadvantage is that tightly coupled devices on the existing network must be transferred simultaneously to the new architecture requiring more extensive planning and cost.

A broker is comparable to a router only implemented in software. Analogous to choosing the type of routers, e.g., Cisco, and designing the physical network between the devices, the first thing that should be decided is what types of brokers, e.g., HiveMQ, and to determine between which devices data will flow. Based on this initial study, the necessary technology to achieve message flow can be implemented in suitable locations. Downtime is costly and can negatively affect a company's reputation, and as a result, redundancy is an essential aspect of all systems. It is achieved in UNS by extensions for broker-to-broker communication, which can be used not only to create redundant systems but as a tool to expand the system when the enterprise grows. It allows for starting locally with brokers run on private servers not connected to the internet. Bridge extensions ensure that expanding the system to include traffic on the internet is not problematic since there are no modifications to the local structure. There are obvious advantages to starting small, such as better control over your own system, reduced initial cost, and the fact that you have not exposed your entire system to hackers on the internet in the initial phase where you are learning and setting up the structure. Broker implementations are based on security protocols like TLS and are commonly developed by companies taking care of all the complex functionally required, like redundancy and reliability. Configuring these products does not require the same level of technical understanding, making them suitable for local deployments by your own personnel.

Successful implementation and the principle behind UNS is the decoupling of devices in the system, thus becoming the strategy for connecting new units. The same strategy can be used as a basis when existing devices are connected to the newly established structure. It is possible to selectively choose which devices are gradually moved over to MQTT structure based on the cost and the value data from these devices have to supervisory or analytical systems by having implemented the broker in its entirety without affecting the existing system. In this way, the parallel system can be maintained as long as it is appropriate, possibly indefinitely.

Not all devices are compatible with MQTT or the publish and subscribe architecture and cannot be integrated directly. These must therefore be connected using intermediary nodes, often called gateways. Gateways are often software modules installed on hosts on the same local network as the edge device. Alternatively, newer PLCs with integrated MQTT drivers can be used as a hub to collect information from various legacy devices and publish on behalf of everyone else to UNS. This is a widely used solution for connecting systems today where a PLC is the single point of communication to a SCADA host interfacing with other PLCs and legacy systems such as ModBus, Profibus, etc. A weakness that we have seen with some of the devices with built-in MQTT drivers is how data sometimes has to be flattened and thus loses some of its value. Consequently, the sender and the receiver have to develop together to agree on a common structure for the communicated data, which is not true to the decoupling principle. An advantage when using purpose-built gateways is that they can structure the information and add scaling or new attributes such as units and ranges, before publishing it to the UNS. Gateways can increase data quality while being faithful to the single source of truth principle by adding the metadata before data is distributed, thereby avoiding the risk of duplicated or inconsistent data due to other systems like data hubs, described shortly, attempting to perform this task.

Data will most likely not become plug-and-play for a long time due to the individual requirements or initiatives of manufacturers and active community individuals. Much of the time to connect systems is often spent on addressing and setting the communication parameters. It is not unreasonable to claim that the most valuable tool for reducing integration costs is online tools such as auto-discovery, browsing, and configuration. It can be achieved with measures such as connection to an endpoint being continuous online, such as a data broker in the cloud.

OPC has attempted to simplify the process by allowing clients to go online on OPC server endpoints to do the configuration while browsing the address space. Consequently, it is common to send fully configured equipment between suppliers so that it can be connected in the lab for online configuration and troubleshooting data sharing before finally being deployed in the field. SparkplugB has attempted to develop plug-and-play for communication over MQTT. Based on information in BIRTH messages, achieving the same structure on the address space while browsing as OPC UA devices is possible. The downside as of now is that few implement the standard. The fact that MQTT devices only connect to the data broker, a common endpoint for all clients, means that communication parameters do not vary to the same extent as before. The broker can be publicly hosted during development at one of the parties or in the cloud, always online, making it possible for the parties to connect with each other from their locations. Another advantage is that the systems are decoupled in time, meaning they can be unit commissioned in any order. It is then of course an advantage if the device producing data is commissioned before the data consumer. Based on this, knowledge and setting up a data broker is something everyone who wants to bet on MQTT as a solution for UNS should set up at their location.

6.3 Security

The security of encryption and authentication algorithms is in itself very secure. It is based on cryptographic keys with a length that is extended to achieve the desired security level or to keep up with developments in computing efficiency and improved algorithms used in brute-force attacks on encrypted messages. The biggest weakness, however, is the human factors that can create security holes, providing hackers access to the systems. The risk of someone accidentally having caused a security weakness will increase as the system gets larger because more people get involved, which is the whole idea behind the UNS. This argues that it is appropriate for brokers to be managed by a small number of qualified personnel, as is the norm in most companies. Alternatively, it can be outsourced to external cloud providers with advantages and disadvantages like trust issues that this entails.

Accidental errors during handshake procedures, storing sensitive information in memory, or irresponsible storage and distribution of certificates can easily happen without a solid understanding of cyber security. Since a lack of understanding is the most common reason for security compromises, not everyone should be able to implement functionality directly related to cybersecurity. Using standardized and proven protocols such as TLS, which the MQTT architecture is based on, and software developed by companies with this sole purpose is an efficient method of minimizing such risk and, thereby the recommended strategy.

Fortunately, the UNS architecture is flexible, allowing the system designer to implement an architecture that considers cyber security challenges. The principle of isolating the system by having no physical link to the internet can be achieved with an MQTT broker architecture. Brokers can be contained to local networks, only enabling internal data sharing within the company. Expanding the network between different sites can be achieved using VPNs connecting brokers to brokers. VPNs have been deployed for site-to-site communications for decades and are a well-tested and trusted technology. The advantages of a VPN are the simplicity of encrypting all traffic at the IP level, being transparent to layers higher up in the OSI model resulting in application data being encrypted without these protocols implementing cryptographic algorithms themselves. Another advantage is that VPNs can be administered by dedicated personnel with a high level of training in cyber security, minimizing the risk of unidentified or unintended security weaknesses.

A good practice is to start with the least amount of access if the architecture is exposed on the internet. In practice, it means that everything is initially locked down, avoiding unknown entry points that hackers can exploit. It is then simpler to create and manage client certificates and open ports and IPs on a need-to basis. The creation and distribution of certificates is a particularly important source of compromise. It can often be cumbersome to distribute certificates, and simple solutions such as someone sending the certificate unsecured by e-mail must be expected. Because certificates are not restricted from being used by several clients, a hacker might gain access to the system by eavesdropping or by a man-in-the-middle attack intercepting and manipulating the email. Although in principle it can easily be detected by keeping track of the IP addresses using specific certificates, it is not something that we see implemented regularly. A solution to minimize the risk is for each site to establish its own root CA that can generate certificates that are verified by the counterpart via the CA chain. Only the root CA certificate needs to be distributed remotely. Then all the client certificates can be created and distributed locally, avoiding the risk when transferring over a global network.

Authentication is an effective tool to prevent outsiders from accessing the system. But as the system grows, gaining more users, it can become problematic if all the users have access to all the data that flows through the network. Therefore, access control by authorization is absolutely necessary and should be implemented at most levels in the hierarchy. PLCs at the lowest level in the architectures are configured according to what is necessary and safe to expose by implementing appropriate read and write protections limiting unauthorized access. Intermediary devices like gateways can further restrict access to variables by setting read and write protections on Tag sets, never violating what has been configured on the edge devices. Role-based authorization to restrict access to selected topics is often implemented in brokers efficiently disclosing information only to intended clients. Finally, physical access to the network can be restricted by routers, switches, and firewalls implementing filtering based on white-listing of allowed addresses or complete isolation. It must be emphasized the ability to individually set access levels without controlling the overarching system that routes data is a necessity for the feasibility of connecting edge devices to the architecture.

6.4 Databases optimization

Initially, we set up a simple and user-friendly database solution. PostgreSQL was an obvious choice for us, as we had experience with it from previous courses. While it was suitable for testing and our specific use case, it is generally not recommended to be used with something as information-heavy as storing everything in the UNS due to its limited scalability. Our database could have been improved by implementing connection pooling to optimize performance and reduce the number of connections needed. This would maintain a set of open connections to our database rather than destroying and creating new ones for each request. Nonetheless, this would only serve as a temporary solution until the number of messages received per second further increased. Multiple databases and filtering that limits the amount of data to each database are necessary when the systems become as large as UNS.

Relational databases like PostgreSQL face various challenges and limitations when storing large amounts of data. These databases must be scaled vertically, requiring more powerful hardware to handle increased data volume and concurrent requests. In contrast, NoSQL databases like MongoDB achieve better scalability through parallel scaling with replication and data distribution. Replication ensures that multiple nodes are available for data access, preventing a single overloaded node from causing extended wait times. Distributing data across multiple nodes allows these databases to accommodate a higher number of requests and concurrent users.

Databases and the amount of information they can hold vary significantly. Vertically scaling databases can only handle so much data before their performance is affected, as increasing the hardware capacity has its limitations. Horizontally scaling databases can generally hold more data as they can distribute it across multiple nodes. However, even these databases may eventually experience performance issues if they grow too large. Every sort of database system has a tolerance that should not be exceeded, and in the case of our project and PostgreSQL, that tolerance is too low if it were to be used in a real production environment.

This would be clearly visible when receiving high rates of incoming messages. After only a short time, the database would hold millions of entries. Even with proper optimization and indexing, the search times for a single entry in this database would be considerably longer than a MongoDB database that uses the same optimization techniques. MongoDB also has multiple procedures that may be employed to further optimize the database.

6.5 Unified namespace possibilities

Much of the motivation behind UNS is to establish connectivity between IT and OT systems, allowing cloud-based applications to access data from the factory floor. Although this chapter does not directly elaborate on the findings of our bachelor's thesis, it is relevant to include it as a feasibility study to showcase the potential opportunities that UNS can enable.

6.5.1 Machine learning

Machine learning truly shines when dealing with massive amounts of data or complex relationships that are difficult to detect or exploit through explicit programming. For instance, ML can be leveraged to optimize controller parameters using time-series data from the dynamic response in the water rig simulator at HVL. The availability of real-time data also allows for continuous monitoring of controller performance, which could degrade because of changing process dynamics caused by component wear. An ML algorithm can then indicate a warning or online adapt the parameters as needed. Moreover, ML can identify hidden patterns that may serve as early signs of wear and the subsequent need for maintenance. Automatic trend monitoring and detecting patterns involving the interplay of several variables are more complex than traditional methods, such as evaluating single measurements, e.g. vibration measurements against predetermined alarm limits can be exploited by ML. The wealth of data made available by UNS serves as both the motivation for implementing and the opportunity to exploit ML's potential.

Common in UNS, data context plays a crucial role in the effectiveness of these systems. In industrial applications, good data is characterized by its relevance, accuracy, consistency, completeness, timeliness, and granularity. Ensuring that data meets these criteria will also significantly enhance the accuracy of machine learning and AI models in generating reliable forecasts and valuable insights.

The field of machine learning is extensive and requires a deep understanding to fully harness its potential. Hence, it is advantageous that providers like Microsoft Azure offer solutions that can be used to implement machine learning or AI algorithms. Like most other platforms, whether it is AWS SageMaker or Google Cloud Platform, Microsoft ML integrates seamlessly with other Azure services, such as Azure IoT Hub and Azure Stream Analytics. It has a wide range of pre-built ML models, pipelines and AI services already available for building, training and supports various deployment methods. The first point of integration would involve connecting the MQTT data to Azure IoT Hub, which can handle large-scale ingestion of device-to-cloud telemetry data. This connection will enable the collection and preprocessing of data from PLCs and other devices within UNS. Azure IoT Hub can then forward the preprocessed data to Azure ML for further analysis.

Azure Stream Analytics, an event processing engine, can analyze real-time device data. It can filter, aggregate and transform the incoming data, allowing us to focus on relevant information and minimize noise in our dataset. Integrating Azure Stream Analytics with Azure Machine Learning enables the use of machine learning models in real-time, potentially predicting errors or anomalies before they occur.

Overall, by integrating Microsoft Azure Machine Learning, IoT Hub, Stream Analytics and integrated with a database like Azure Cosmos DB provides a robust and scalable system that leverages machine learning and AI to analyze UNS data, predict errors and provide valuable insights into our connected devices and processes. However, like most cloud platforms, it can become expensive depending on the scale and complexity of the project.

6.5.2 Information hubs

Data has little value if it is not adequately structured or labeled according to its production. Within a UNS architecture, numerous devices connected to the cluster may generate vast amounts of data that need to be managed, processed, and analyzed effectively. One key challenge is the process of standardizing and contextualizing the data to provide the best possible understanding and visualization for future decision-making dependent on the data.

Information hubs serve as middleware solutions that connect to various data sources, including PLCs, sensors, and other IIoT devices. These hubs can receive messages in various formats and process the data to generate a standardized and contextualized format before it is published or stored in a database. By doing so, they enable seamless data integration, facilitate advanced analytics, and provide real-time analytics support. Information hubs play a crucial role in addressing these challenges by providing a centralized data collection, management and processing platform.

HighByte is a solution aiming to standardize and contextualize industrial data. It is comparable to Ignition Gateway, except that HighByte works with data from a collection of PLCs and other data-producing devices. It is a central hub for transforming data into specific data-modeling schemas for all other consuming applications. Standardization reduces the time and effort required by other systems to preprocess the data. Unlike being focused solely on MQTT as a means for data sharing, HighByte integrates with various servers and APIs, truly making it a solution for the concept of a unified namespace.

6.5.3 MES

In today's business landscape, where time-to-delivery must be minimized and companies strive to avoid excess inventory to save cost and avoid product expiration, the ability to adapt quickly is crucial. To be able to make quick but wise decisions, one is dependent on good information. An overview of the plant's overall status, or the status of multiple plants, is often necessary for effective resource and manufacturing schedule planning. Manufacturing and Execution Systems (MES) are systems developed to centralize the entire production at the management levels. These systems utilize planning tools that integrate production resources, vendor deliveries, and customer demands to optimize the production schedule. Information from MES systems is also valuable for strategic decision-making, such as investment considerations by enterprise owners. Traksys is a MES system that intends to deliver real-time actionable productivity intelligence which can be used to rapidly adapt based on the overall status of the enterprise and external demand.

6.6 Future work

The UNS poses a challenge to the well-established security concepts companies have implemented to secure their data. To further explore this topic, it would be beneficial to examine best practices in the cybersecurity domain when transitioning from segmented architectures to a more open architecture, having a larger attack surface and an increased number of connected users.

In addition, it may be valuable to explore the topics of auto-discovery and online tools like reflection or inference [40], as they can significantly reduce integration times and increase profitability.

Redeploying the UNS broker solution on an AMD hardware infrastructure capable of supporting Kubernetes with DNS discovery will enable the cluster broker to become a highly scalable solution, similar to our desired solution described in Chapter 4.1.1.6.

7 Conclusion

During our thesis, we have developed a system that utilizes redundant HiveMQ brokers hosted both locally and in the cloud to connect a wide range of IT and OT clients, including PLCs, OPC server gateways, smart sensors, SCADA systems, databases, and a dashboard. These clients serve different purposes, but they all communicate through event-based publishing and subscribing to minimize traffic while maintaining device state awareness in the system. We see few alternatives apart from MQTT when looking for technologies that can be used to implement the concept of a Unified Namespace. At this point, it seems to be more of a question of implementing the MQTT architecture or not transitioning to UNS at all. One major advantage of MQTT is that it can coexist on the same networks as all other devices in the enterprise, not requiring all devices to support its technology. Consequently, adopting a UNS strategy realized with the MQTT broker architecture does not require an all-or-nothing investment since it can be implemented on existing Ethernet infrastructure.

The advantages of MQTT include its flexibility and high scalability, which make it ideal for integrating new devices under the Industrial Internet of Things (IoT) category. As demonstrated in our thesis, MQTT provides a relatively simple way to integrate these devices. While ad hoc integrations may not always be the most effective, MQTT also enables solutions for optimized channels between different systems, such as a SCADA system and an OPC server, using technologies like SparkplugB. Although our tests generally show fast response times, we have encountered incidents where latency was high. These results are typical for Ethernet, which is a best-effort protocol. While SCADA may accept this variability as a supervisory system, not all industrial controllers can, and they need to be located in deterministic networks. However, this does not mean that these controllers cannot share relevant data with a UNS architecture.

Data organization is more crucial than ever when gathering everything in a unified namespace. A practical method of naming objects is to use physical location, considered safe and scalable since two things cannot occupy the same space. It is crucial to have a sufficiently deep hierarchy like ISA-95, which is publicly accepted and can accommodate enterprises spread out over multiple locations. In addition to hierarchical naming, standardized metadata is tremendously valuable, providing context that enables automatic scaling, alarming, and other processing functions by the receiving systems.

Alternatively, to standardized metadata is to infer the information structure during operation, commonly called reflection in IT systems. Plug-and-play data integration may still be challenging, but online tools such as auto-discovery and browsing can undoubtedly help minimize integration times by online inference. It is partially the success behind the OPC UA standard and something that MQTT is trying to add to networks where both IT and OT devices are connected, helped along the way by initiatives such as SparkplugB. Although these initiatives are still in their early stages and require a commitment to specific protocols or vendors, they can potentially become the new norm for automatic sharing in industrial networks.

The consequences of losing control of one's data can be catastrophic since then others can manipulate the system, sensitive company secrets can be compromised, or the systems can be locked using encryption by hackers. It is therefore understandable that owners may be reluctant to connect their entire company infrastructure to the Internet. It is then an advantage that MQTT brokers work just as well on local networks and can easily be expanded later, e.g., bridged between sites using VPNs without unnecessary and avoidable costs.

APPENDIX A - Reference list

[1] HiveMQ GmbH, "HiveMQ Documentation," 2023. [Online]. Available: [Accessed: Feb.5, 2023].

[2] A. Noah, "Vector Logo Zone - Gorgeous SVG logos," 2023. [Online]. Available: https://www.vectorlogo.zone/. [Accessed: Apr. 3, 2023].

[3] ChatGPT, "chatGPT logo," 2023. [Online]. Available: https://en.wikipedia.org/wiki/ChatGPT#/media/File:ChatGPT_logo.svg [Accessed: Apr. 23, 2023].

[4] Siemens, "OPC UA PubSub with SIMATIC S7-1500 based on MQTT" 2021. [Online]. Available: https://cache.industry.siemens.com/dl/files/826/109797826/att_1085877/v3/109797826_OPC_UA_PubSub_MQTT_DOC_V1_0_en.pdf[Accessed: Mar. 03, 2023].

[5] Vonnegut, "The CIA information security triad," 2016. [Online]. Available: https://devopedia.org/information-security-principles [Accessed: Jan. 23, 2023].

[6] *Sparkplug Specification Project Team, "Sparkplug 3.0.0: Sparkplug Specification," Eclipse Foundation, 2022 [Online]. Available: https://sparkplug.eclipse.org/specification/ [Accessed: Nov. 16, 2022.].*

[7] P. Schume, "What is Industry 4.0 and how does it work? | IBM," 2023 [Online]. Available: https://www.ibm.com/topics/industry-4-0 [Accessed: Jan. 28, 2023].

[8] M. Sen Gupta, "What is Digitization, Digitalization, and Digital Transformation?," ARC Advisory group, 2020 [Online]. Available: https://www.arcweb.com/blog/what-digitization-digitalization-digital-transformation [Accessed: Jan. 23, 2023].

[9] Skai, "How to Use Customer Behavior Data to Improve Your Marketing Strategy," Mar. 24, 2022 [Online]. Available: https://skai.io/blog/customer-behavior-data/ [Accessed: Jan. 25, 2023].

[10] F. Sofio, "How Machine Learning is Helping Engineers with Predictive Maintenance and Prevent Equipment Failures | Valispace," @vali_space, Jan. 31, 2023 [Online]. Available: https://www.valispace.com/how-machine-learning-is-helping-engineers-with-predictive-maintenance-and-prevent-equipment-failures/ [Accessed: Feb. 18, 2023].

[11] Cisco, "How Is OT Different From IT? OT vs. IT," 2023. [Online]. Available: https://www.cisco.com/c/en/us/solutions/internet-of-things/what-is-ot-vs-it.html [Accessed: Jan. 20, 2023].

[12] S. J. Bigelow and B. Lutkevich, "What is IT/OT convergence? Everything you need to know," TechTarget, Aug. 2021. [Online]. Available:

https://www.techtarget.com/searchitoperations/definition/IT-OT-convergence [Accessed: May. 12, 2023].

[13] S. Gannu, "Standardize OT Device Drivers At The Edge," Forbes, Jan. 13, 2020 [Online]. Available: https://www.forbes.com/sites/forbestechcouncil/2020/01/13/standardize-ot-device-drivers-at-the-edge/ [Accessed: Feb. 1, 2023].

[14] M. Widjaja, "Decoupling Architecture," @InfoItArch, 2023. [Online]. Available:https://www.itarch.info/2020/05/decoupling-architecture.html [Accessed: Jan. 12, 2023].

[15] M. Parris, "How all protocols fail at data access interoperability," Industrial Ethernet Book, vol. November, p.34-45, 2022. [Online]. Available: https://iebmedia.com/ebooks/november-2022-industrial-ethernet-book/ [Accessed: Jan. 3, 2023].

[16] N. Yau, "Understanding Data - Context," Big Think, 2013. [Online]. Available: https://bigthink.com/articles/understanding-data-context/ [Accessed: Jan. 6, 2023].

[17] Microsoft, "Namespaces (C++)," 2021 [Online]. Available: https://learn.microsoft.com/en-us/cpp/cpp/namespaces-cpp?view=msvc-170 [Accessed: Jan. 10, 2023].

[18] J. Hottell, "Efficient IIoT Communications," Cirrus link, pp. 48, 2019. [Online]. Available: https://cirrus-link.com/wp-content/uploads/2019/12/Efficient-IIoT-Communications.pdf [Accessed: Jan.10, 2023].

[19] Tripwire, "Cybersecurity," TripWire's Editorial Staff, 2019. [Online]. Available: https://www.tripwire.com/state-of-security/air-gap-industrial-control-networks [Accessed: Jun. 17, 2023].

[20] *OPC Unified Architecture*, **IEC62541**, O. Foundation, 2020.

[21] *MQTT Version 5.0*, **ISO/IEC 20922**, O. Standard, 2019. [Online]. Available: Latest version: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[22] *ISA95, Enterprise-Control System Integration*, **IEC/ISO 62264**, ISA, 2016.

[23] W. Stallings, *Cryptography and Network Security Principles and Practice*, 7 ed. Global edition: Pearson Education Limited, 2016.

[24] "Virtual private network - Wikipedia," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Virtual_private_network [Accessed: Feb. 5, 2023].

[25] GeeksForGeeks, "Web Technology," 2023. [Online]. Available: https://www.geeksforgeeks.org/web-technology/ [Accessed: Jan. 22, 2023].

[26] A. M. Kurian, "Understanding Protocol Buffers," in *Better programming*, 2020. [Online]. Available:

https://betterprogramming.pub/understanding-protocol-buffers-43c5bced0d47 [Accessed: Apr.22, 2023].
[27] N. Ramirez, "Load Balance Your Servers - HAProxy Technologies" Jun. 05, 2021. [Online] Available: https://www.haproxy.com/blog/haproxy-configuration-basics-load-balance-your-servers/ [Accessed: Feb. 04, 2023]
[28] The Linux Foundation, "Kubernetes Overview," 2023 [Online]. Available: https://kubernetes.io/docs/concepts/overview/ [Accessed: Feb. 21, 2023].
[29] Docker Inc., "Docker containers," Nov. 11, 2021. [Online]. Available: https://www.docker.com/resources/what-container/ [Accessed: Feb. 7, 2023].
[30] HiveMQ, "HiveMQ Media Kit - High Resolution Logo Downloads," 2023. [Online]. Available: https://www.hivemq.com/logos-and-media-kit/ [Accessed: Apr. 19, 2023].
[31] Standards Norway, "NORSOK," **I-005**, 2021. [Online]. Available: https://www.standard.no/en/sectors/energi-og-klima/petroleum/norsok-standards/#.Y_cw72nMK3A [Accessed: Feb. 19, 2023].
[32] O. Sande, *"Sikkerhet, Styring/regulering og Overvåkning"*, Western University of Applied Science: Olav Sande, 2020, p. 79.
[33] OPC Foundation, "UA Companion Specifications," 2023. [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua/ua-companion-specifications/ [Accessed: Feb. 12, 2023].
[34] Espressif Systems, "ESP32-Ethernet-Kit V1.2 Getting Started Guide - ESP32 - ESP-IDF Programming Guide latest documentation" Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-ethernet-kit.html [Accessed: JFebn. 10 6, 2023].
[35] Espressif Systems, "ESP32-DevKitC V4 Getting Started Guide - ESP32 - ESP-IDF Programming Guide latest documentation". 2023 [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html [Accessed: Jan. 10, 2023].
[36] P. Van Oosterwijck, "wESP32: Wired ESP32 with Ethernet and PoE," hackadayio, Sept. 30, 2021. [Online]. Available: https://hackaday.io/project/85389-wesp32-wired-esp32-with-ethernet-and-poe [Accessed: Jan. 6 2023].
[37] Sonatype, "Maven central library," Sonatype, Inc., Available: https://central.sonatype.com [Accessed: Feb. 2, 2023].
[38] Wikipedia Foundation, "Algorithms for calculating variance," [Online]. Available:

https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance [Accessed: Feb. 2, 2023].

[39] IBM, "Numbers - IBM Documentation," Feb. 7, 2023. [Online]. Available: http://www.ibm.com/docs/en/idr/11.4.0?topic=types-numbers [Accessed: Feb. 13, 2023].

[40] Microsoft, "Reflection in .NET," Sept. 15, 2021. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection [Accessed: Feb. 3, 2023].

[41] E. Systems. "ESP32-WROVER-E & ESP32-WROVER-IE Datasheet." [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-wrover-e_esp32-wrover-ie_datasheet_en.pdf [Accessed: Jan 10, 2023].

[42] L. Technologies. "LTC4267 Power over Ethernet IEEE 802.3af PD Interface with Integrated Switching Regulator DATASHEET." [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/4267fc.pdf [Accessed: Jan 7, 2023].

[43] L. Woodahl. "Ethernet PHY PCB Desgin Guidelines." [Online]. Available: https://www.ti.com/lit/an/snla387/snla387.pdf [Accessed: Jan 6, 2023].

[44] Microchip. "LAN8720A Datasheet." [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/8720a.pdf [Accessed: Jan 6, 2023].

[45] S. LABS. "SINGLE-CHIP USB-TO-UART BRIDGE CP2102/9 Datasheet." https://www.silabs.com/documents/public/data-sheets/CP2102-9.pdf [Accessed: Jan 6, 2023].

[46] A. LaMothe, "Crash Course Electronics and PCB Design," Udemy, pp. [Online]. Available: https://www.udemy.com/course/crash-course-electronics-and-pcb-design/ [Accessed: Dec 12. 2021].

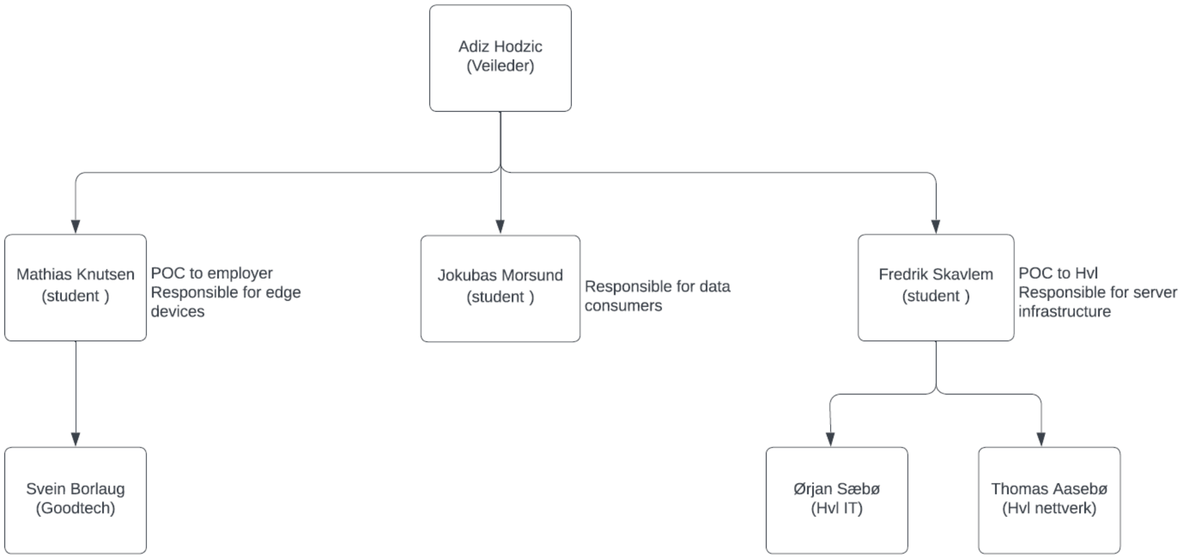
[47] P. Salmony, "Phil's Labs Videos," [Online]. Available: https://www.phils-lab.net/videos [Accessed: Jan 3, 2023].

[48] R. Feranec, "How to Decide on Your PCB Layer Ordering, Pouring and Stackup (with Rick Hartley)," Mar. 4, 2021. [Online]. Available: https://www.youtube.com/watch?v=52fxuRGifLU [Accessed: Jan. 8, 2023].

[49] B. Suppanz, "Printed Circuit Board Trace Width Tool | Advanced Circuits," [Online] . https://www.4pcb.com/trace-width-calculator.html [Accessed: Jan 8. 2023].

[50] Siemens, "Programming an OPC UA .NET Client with C#" [Online]. https://support.industry.siemens.com/cs/document/42014088/programming-an-opc-ua-net-client-with-c-for-the-simatic-net-opc-ua-server?dti=0&lc=en-DZ [Accessed: Jan 20. 2023].

APPENDIX B - Project organization



The project group consists of three students from Western University of Applied Science and are under the supervision of college lecture Adis Hodzic.

Work tasks are clearly divided between the group members. Cross-coordination to integrate subsystems are done by weekly meeting and when the need arises.

We have established weekly coordination meetings which are conducted over microsoft teams:

- Week start - Monday 10:00 (every week)
- Internal coordination - Thursday 1100 (every two weeks)
- Supervisor coordination - Thursday 11:00 (every two weeks)

Task tracking, document control and project organization are administered through teams. Source control is by GitHub.

Responsibilities for external coordination has been distributed to ensure continuity.

8 APPENDIX C - Design and manufacture of UNS printed circuit board

For our thesis, we wanted to apply our knowledge of electrical engineering disciplines as well. Therefor we designed, built, and tested two PCBs during the course of this thesis. This appendix will cover these aspects.

8.1 Overarching design philosophy

The entire PCB is divided into four quadrants. The philosophy behind this idea is to enable independent debugging of each section. Each section is separated by a jumper bridge, which acts as a quadrant isolator. As the PCB can have multiple potential power sources, we incorporated an onboard voltage regulator and used diodes to isolate each supplier, considering the voltage span of 5V to 12V from the various sources. This regulator provides the PCB with 3.3V DC and a built-in LED for easy determination of power status.

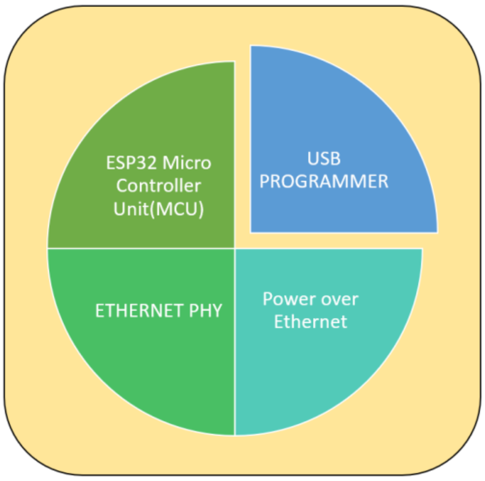


Figure 89 Segregated PCB Quadrants

8.1.1 ECAD

ECAD, or Electronic Computer-Aided Design, is a crucial component of modern electronic design and engineering. It encompasses the use of software tools to create, design, simulate, and analyze electronic circuits and systems. ECAD software allows engineers to efficiently design printed circuit boards (PCBs), schematics, and various electronic components.

The software we used to design the PCB is EASYEDA, primarily because of previous experience with the software. Although Altium Designer is considered the best software for PCB design, it was not chosen due to its high subscription fees. As a result, we had to manually route all the individual traces on the board, with limited automation available for tasks like IC fan-out and auto tracing.

EASYEDA consists of two designers. The first designer is used to create the actual circuit we intend to build in a schematic format, as shown in Figure 88. The second designer is utilized for tracing the individual traces on the PCB based on the circuit schematic drawn, as depicted in Figure 87.

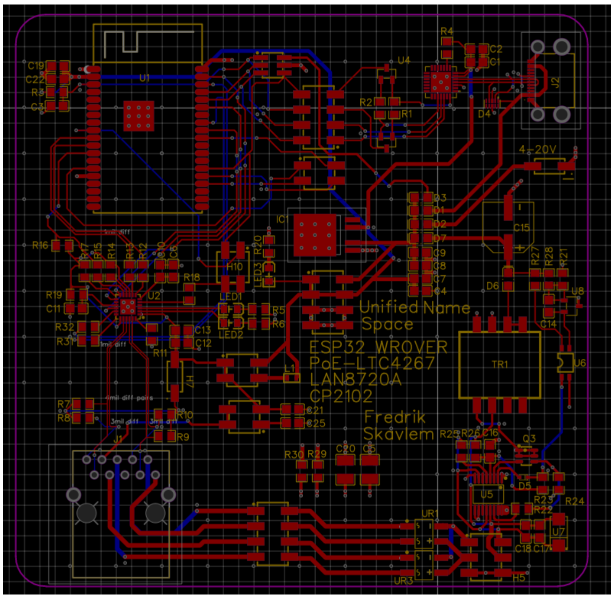


Figure 90 prototype trace layout without ground-planes, red toplayer, blue bottomlayer

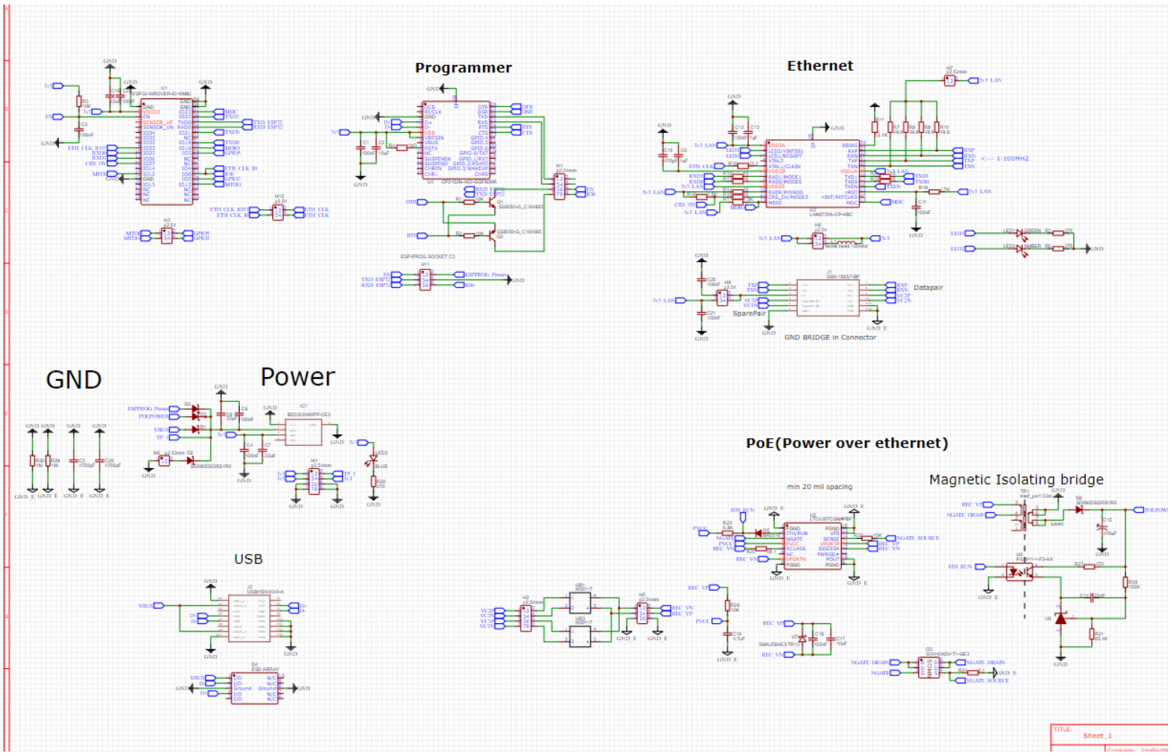


Figure 91 Initial protoype schematic design

8.1.2 MCU

The initial component chosen was the MCU (Microcontroller Unit), which needed to support multiple programming languages such as C++, Python, and C# if possible. We decided to base our design on the ESP32 WROVER-E System on Chip (SoC). This particular SoC has the potential to run micropython, nanoFramework (C#), and C++. Additionally, it offers built-in Bluetooth and Wi-Fi capabilities. We selected the WROVER N16R8-E due to its ability to run two threads in parallel and its built-in 16MB flash memory and 8MB usable PSRAM. Design reference found in datasheet [41]

8.1.3 Power over Ethernet

The module should have the capability to run on the industry-standard Power over Ethernet (PoE), which enables powering the entire PCB through the supplied current via an Ethernet cable. Currently, there are multiple PoE protocols available that can deliver power ranging from 15.4W to 100W. For our purposes, we will focus on the 15.4W protocol for two reasons. Firstly, it simplifies the PCB build by requiring fewer components, and secondly, it prioritizes safety.

The 15.4W protocol can be further categorized into passive and active components. The active components are controlled by the PoE injector, which supplies power through the Ethernet cable. The PCB must support both passive and active protocols, which is why the IC LTC 4267 [42] was chosen to accommodate the active protocol for receiving PoE. Passive support does not require any power handshake to initiate power delivery.

Initially, there was consideration to layout the PoE module on a separate PCB. However, due to the cost-effectiveness of ordering a PCB with a small size, we opted to integrate it into the main board with a separate ground. For the final design, we will reassess the separation of the PoE module, ensuring a minimum of 2.54mm space between the transformer and neighboring ICs on the board, in accordance with Texas Industries PCB guidelines [43].

8.1.4 LAN

Since our goal was to incorporate PoE capability into the PCB, it made logical sense to include Ethernet LAN support as well, since we needed the ethernet jack for PoE. Initially, we opted for the LAN8720A due to its usage in another design alongside the ESP32 WROVER (https://github.com/c-/ESP32-Ethgate), which intrigued us because of its minimal component requirements. However, upon closer examination of the LAN8720A datasheet [44], we discovered that the manufacturer recommended additional resistors and capacitors to be added to the IC. For the prototype, we decided to follow the manufacturer's recommendations. However, we will utilize the prototype to assess the feasibility of removing some of these resistors without experiencing packet loss in TCP/IP traffic.

8.1.5 Programmer

The module should be programable without using external Programmer hardware, it should also be connectable by the EU standard USB-C. In order to achieve this, we chose the CP2102 USB to UART bridge. Initially FT2232H was chosen, but due to no need for the onboard debugger JTAG, it was discarded due to higher cost than the CP2102. Datasheet used during the design phase to ensure values for RC delay on data lines other various design criteria [45].

8.1.6 Debugging

In order to increase our success rate with the prototype, due to our limited PCB manufacturing experience. We attempted to identify the areas in the circuit design that may have uncertainties. This is where the different datasheets come into play, as each one provides a recommended circuit application for the interconnected ICs in the build. To avoid being locked into a single solution, we added several pin headers to the design. This allows us to reconfigure the board using wires during the testing phase of the manufactured board. However, it's important to note that certain critical data lines are protected from this practice to ensure their integrity. The impact of adding pin headers to the traces, which can alter the trace impedance, will be discussed in Chapter 10.2.4 Trace routing.

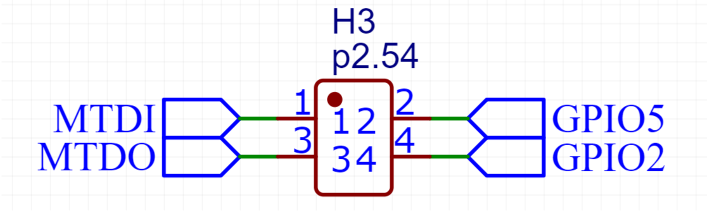


Figure 92 Example of the use of pinheader on strapping pins

Additionally, we took the step of separating the various ICs on the board so that not all modules will be operational when the card is connected, unless a jumper is added to the pin header to supply power to the specific IC.

8.2 PCB layout design criteria's of prototype

One of the most important elements when designing a PCB is the always striving for reduction of electromagnetic interference. In this design we have followed multiple guidelines for ensuring as best results as possible, the routing of Ethernet on the PCB is heavily influenced by Texas instruments ethernet PHY PCB design layout checklist [43]. These guidelines give directions on how magnetic isolation, earth ground isolation, differential pair tracing and differential pair should be done.

The design and routing of the PCB is also based on the advices gives trough the e-course Crash Course Electronics and PCB Design [46] and the YouTube channel Phil's lab [47]

8.2.1 Units

In this appendix the unit mil will be used which refers to one thousandth of an inch, this is due to all documentation used in this build have been using this unit, including EASYEDA. Therefore, it was advantageous for us to change from metric, rather than converting all units. For reference 1mil = 0.0254, 10mil = 0.254mm, 50mil 1.27mm, 100 mil = 2.54mm.

8.2.2 PCB layer stack up and size

A very important step in PCB design is always to try to reduce EMI. One key decision is what layer stack up to choose, this means in practice how many layers should the board consist off and what should be on the different layers. One of the primary drivers of PCB layer stack-up has been price. The rule of thumb is the more layers a board has the more expensive it is. The manufacturer we have chosen to use is JLCPCB, and with a 2-4 layer board the price is 7$, with a 6 layer board the price is 63$. Therefor a 4 layer board has been chosen. We wanted to get the test board as big as possible without major price changes to accommodate as many build possibilities on one board. Therefore a max-limit was set to 95mm x 95mm, as this size will still give a board for 7$

The recommended stack-up is widely debated, one of the more popular 4-layer board stack-up is signal-signal-ground-power. This is a stack we do not wish to use, the reasoning behind is our own intuition on how the electromagnetic field works and the detailed PCB ground plane seminar of Rick Hartley (Principal Engineer at RHartley Enterprises, and pcb designer for last 40 years) [48]

As signal paths will be based on a positive current, the magnetic field will need a return path to the ground plane, this means if we have multiple signal layers stacked on top of each other, the signal layers will reference the ground plane thro the other signal layer. Therefore, we chose to use a signal/power-ground, ground-signal/power that will have a ground reference plane straight under the trace strip. This also has the added benefit of reducing crosstalk between traces on the board as we at certain places have to traces underneath another.

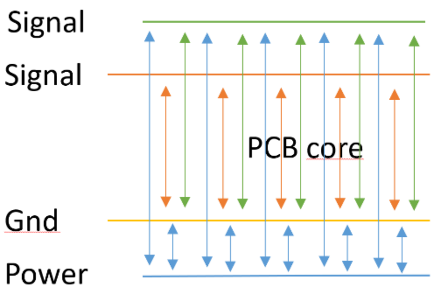


Figure 94 Bad layer stackup, that increases EMI propagation between traces.

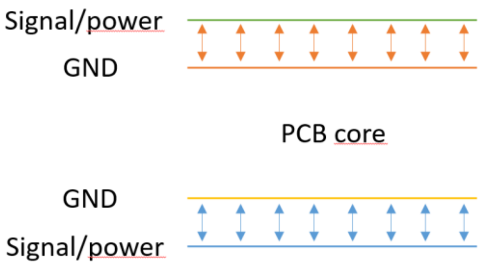


Figure 93 Optimal layer stackup to reduce EMI propegation

The reduction comes from the dual grounding planes in the PCB, making both signal planes reference each other. In order to equalize the 2 ground planes as much as possible we have stitched them together with VIAs(trace tunnel through PCB to connect two traces) the stitching vias are placed with 300 mil spacing where possible trough out the PCB.

8.2.3 Magnetic isolation

When utilizing PoE, two different ground connections will be present on the card. This distinction arises from a variance in ground references between the PoE injector and the PCB, where each may have their own respective ground references. As a result, these two ground planes are separated by a minimum distance of 200 mils. While the TI guidelines [43] recommend a minimum separation of 20 mils, we have incorporated a larger tolerance due to the size of the test PCB.

Furthermore, the guidelines suggest establishing a connection between earth ground and ground using a capacitor and a high-value resistor of 1M ohm. After evaluating various designs, we have opted for a parallel configuration with a 4700pF capacitor, which aligns with our chosen approach as well.

In terms of transformer placement, the guidelines dictate that it should be positioned no less than 2.54 mm away from any other IC on the board. In our design, we have two transformers: one located at TR1 and the second situated within the Ethernet connector.

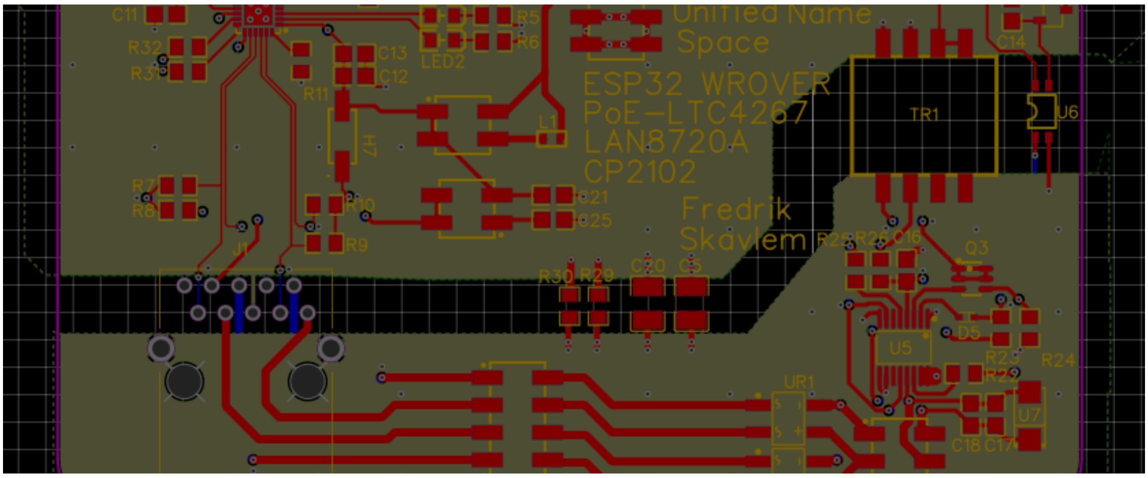


Figure 95 Protype magnetic isolation bridge in groundingplane

8.2.4 Trace routing

The main philosophy when hand routing the traces on the board is to start with component placement. After the components have been placed out with the magnetic specifications distance and components like coupling capacitors for ICs are placed, the first traces are routed. The first to be traced are critical data-lines where differential pairs need to be no more than. In this design the critical data-lines are.

- Differential pair from Ethernet-jack to LAN IC (TXN/TXP and RXN/RXP)
  - Maximum 50mil in difference between differential pair
  - TXN/TXP pair has a total of 3 mil difference.
  - RXN/RXP pair also 3 mil in difference.
- Differential pair LAN IC to ESP32 (TX0/TX1 and RXD0/RXD1)
  - Maximum 50mil in difference between differential pair
  - TX0/TX1 pair has 1 mil difference
  - RXD0/RXD1 pair has 1 mil difference

In accordance with the TI ethernet guidelines [43] the differential impedance of TXN/TXP and RXN/RXP should be 100 ohm, these are called the MDI traces in the guideline. We used EASYEDA impedance calculator and to get the differential impedance at 100 ohm and the traces width must be 6.89mil.

TX0/TX1 and RXD0/RXD1 are called MII traces and should have a single ended 50 ohm impedance. This gives us a recommended trace-width of 11.55mil.

Its recommended to only cross signal paths diagonally when routing PCB. This is ignored due to the stackup of this PCB with double ground layer in 2 center layers. When routing return path should always be considered, when placing a VIA through the board, if possible a ground via should accompany it to reduce EMI. For powerlines we have assumed a maximum of 1 A traveling in the 3.3V powerline. The pcb copper pour is from the manufacturer set to 1 oz/ft. Using the online trace width calculator [49] it will give us a raise of 2.92 celcius at an ambient temperature at 25 celcius using a tracewidth of 25 mils.

8.2.5 Component packages

When building the PCB, the mounted IC's comes in different packages and have 2 over arching charactertics, mainly Surface mounted devices(SMD) the IC only have connection points on one surface, trough hole components that pierces all layers of the board. Using SMD components will greatly reduce the amount of handsoldering needed due to the Reflow soldering technique we will be applying to the PCB.

As previous mentions the IC's comes in different packages. These have specific designators like QFN, SOP, SOT which indicates what type of package they have. We will avoid the BGA package at all costs as this package have leads where we connect to the IC underneath it. This will make it practically impossible for us to detect and correct any soldering flaws on that particular IC.

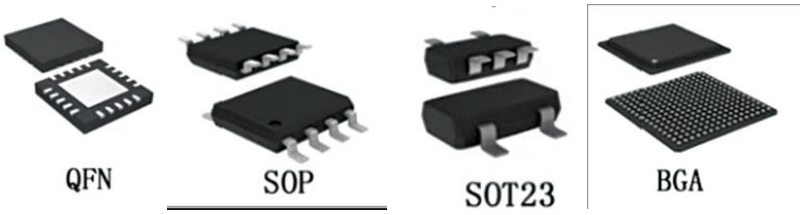


Figure 96 Different IC packages

https://www.electronicsforu.com/wp-contents/uploads/2019/12/1-1.jpg

8.2.6 Soldering techniques

The main soldering technique we will be using is Reflow, this technique is done by applying solder paste with pre-manufactured solder stencil to apply the solder paste to the PCB. Then we will add the components by hand onto the PCB and using a Reflow oven to bake the PCB. The temperature set on the oven follows the particular reflow curve specified in the datasheet of the used solderpaste.

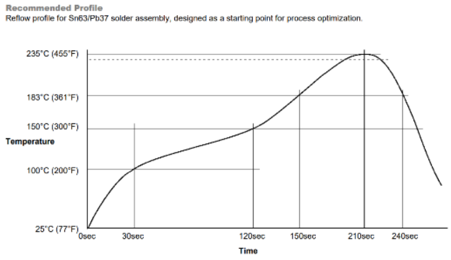


Figure 97 Solder paste reflow curve

Since we are not using additional framework on the SMD stencil, we will accidentally add more solder paste than necessary. Therefor the PCBs will need to be inspected and corrected by hand using heat gun and microscope in combination with solder flux, where the flux is added to ensure no additional oxidation of the solderpaste when remelting with heatgun. We have observed that this phenomena has a tendency to happened with QFN packages.

8.3 Testing regime of prototype

After the soldering of the board, we need to perform a series of tests on the board. We have determined that PoE section have the highest risk of potential destroying the prototype and will therefore be tested last. During the testing a pre-written test program will be used that asks the MCU to send a few characters over the serial link such that we may determine 2 way connection to MCU via UART. Underneath we will list in what order we are to test the prototype board.

1. Evaluate ESP32
    a. Program ESP32 using offboard programmer ESP-PROG device using test program. The offboard programmer will supply the card with power through the 1.27mm header.
    b. no additional power to be added to the card at this point.
    c. blue led onboard should indicate power on board.
    d. if no response from MCU check strapping pins on H3 in accordance with datasheet
        i. ground MTDI.
2. Evaluate onboard USB-UART IC by using onboard USB-C connection from computer.
    a. Ensure pin-headers H1 are jumped before testing.
    b. Load test program from USB-C to determine connection
3. Evaluate LAN module
    a. Add jumper to pinheader H6, H9 and H10 before testing.
        i. H10 ETH_CLK to be connected to ETH_CLK_IO33 at initial try.
        ii. H6 3v3_LAN to be connected to ferrite bead.
        iii. H9 to be connected to capacitor to GND.
    b. Loading a program on MCU, that will try to establish LAN connectivity using ethernet.
    c. Test for packetloss over ethernet to ensure stable connection.
4. Replace R12/R13/R14/R15/R31/R32 with 0 ohm resistor, retest for package loss.
5. Test of PoE.
    a. Ensure board is disconnected from all power sources.
    b. Add jumpers to pinheaders
        i. H2 straight bridge jumpers
        ii. H5 straight bridge jumpers
    c. insert a non-connected ethernet cable to PCB.
    d. BEFORE ADDING POWER. Ensure PCB isolation, as PCB will receive up to 57 volts and due to the voltage, electrical transfer via skin-contact is possible.
    e. Insert PoE injector between LAN switch and PCB ethernet connector.

8.3.1 Prototype 1 Build phase

After receiving the components, we promptly began assembling our PCB design. First, we secured the PCB to the table using tape. Then, we placed the SMD stencil onto the PCB and applied solder paste. Using a plastic card, we carefully spread the paste evenly over the stencil. After a few attempts, we achieved a highly satisfactory result with the solder paste uniformly applied to the PCB.

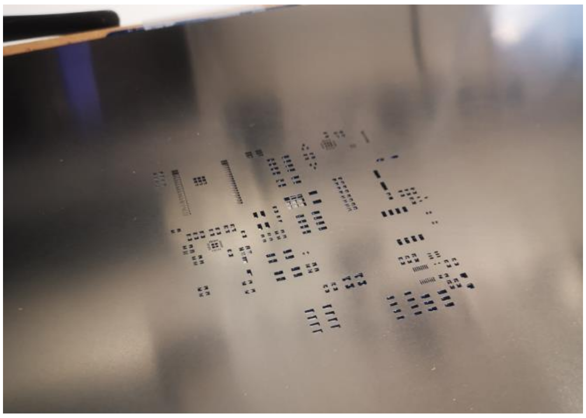


Figure 98 SMD stencil

With the solder paste in place, we proceeded to manually add all the various components to the board. During this process, we encountered our first design mistake. We had mistakenly acquired a MID USB-C connector that had pins located 2mm above the PCB. To address this issue, we bent the connectors at an angle so that they could properly connect with the pads on the PCB.

Next, we placed the PCB in the school reflow oven, which was set to match the reflow profile of the solder paste. However, due to our previous makeshift fix, the USB-C connector did not attach itself correctly to the pads. Consequently, we had to resort to hand soldering the USB-C connector. This presented a significant challenge, as the small size of the USB-C connector's pads (0.2mm spacing) made it impossible to solder them without the aid of a microscope. Fortunately, we were able to utilize the microscope available at the university to assist us. This enabled us to identify any potential short circuits while soldering. After numerous attempts, we successfully soldered the USB-C connector to the PCB.

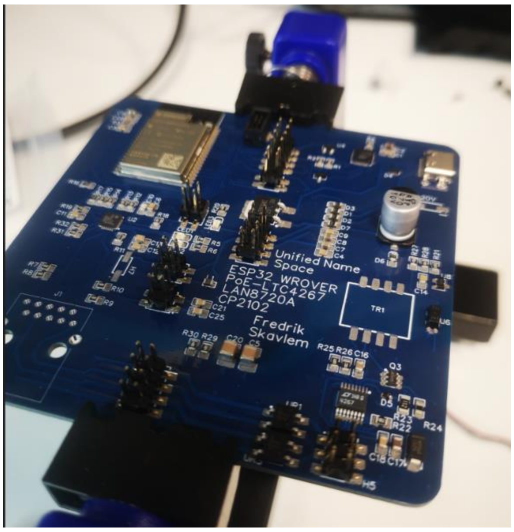


Figure 99 Components placed on solder paste

8.3.2 Initial Power testing

To ensure that our PCB was powered, we began by connecting a 5V source to the power circuit. We immediately noticed that our Blue LED illuminated, indicating that the regulator was functioning properly. To conduct a safe and thorough test, we measured the voltage output from the voltage regulator and confirmed that it measured 3.3V, aligning with the specified value.

8.3.3 Test of ESP32 MCU quadrant

We initially focused on testing the ESP32 quadrant since it was crucial for the operation of the other quadrants. As the other quadrants were disconnected at this stage, we didn't have access to an onboard programmer. Instead, we utilized an offboard programmer, specifically the ESP-PROG with an FT2232HL USB-UART bridge. This choice was primarily based on the group's experience with this particular debugging tool.

We programmed the ESP32 with a small code that would transmit a string through the UART connection. Once the MCU was successfully loaded, no further modifications were required to the quadrant for it to function. The ESP32 promptly began publishing the designated string on the UART connection. No adjustments to the strapping pins were necessary.

8.3.4 Test of USB programmer quadrant

After ensuring that the MCU quadrant was operational, we proceeded to connect the on-board USB programmer quadrant to the MCU quadrant and plugged in a USB-C cable to the PCB. However, initially, nothing happened. We were unable to establish a connection with the CP2102 module from a computer, leading us to believe that the module was not receiving power. Normally, we should be able to detect the module as a COM port on a computer.

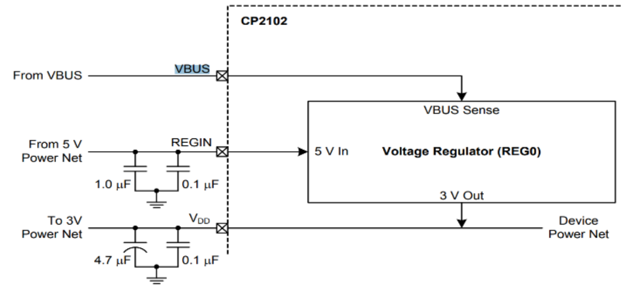


Figure 100 Design recommendation of power on CP2102

Since we had confirmed the functionality of the 3.3V circuit on the board during the previous MCU testing, we investigated further. Upon reviewing our design, we discovered that we had missed a connection from the USB-C 5V line to the VBUS pin.

We resolved the issue temporarily by hand soldering a cable from the 5V output of the regulator. However, it's important to note that this solution will not be implemented in the final design. This is because there may be situations where the input voltage to the regulator exceeds 5V. After implementing this temporary fix, the module powered on successfully, and we were able to detect the CP2102 on our computer as a COM port.

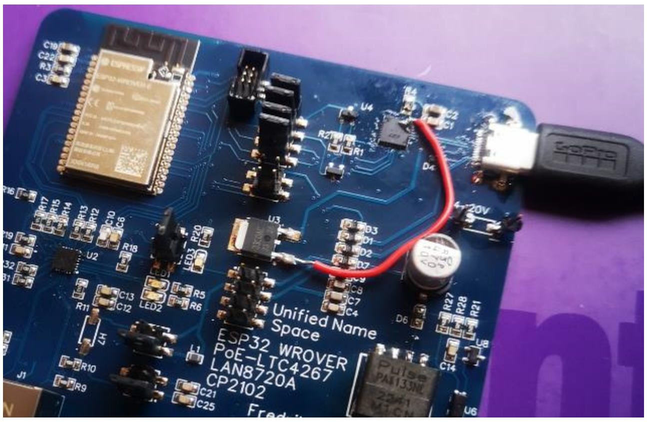


Figure 101 5V to VBUS cable.

When attempting to load the program onto the ESP32 quadrant using the CP2102, we encountered an error stating that the ESP32 was not in download mode. This indicated a problem with our auto-configure circuit, which was based on the recommended schematic for the CP2102. This circuit is responsible for switching the EN and IO0 pins on the ESP32 module to enable download mode.

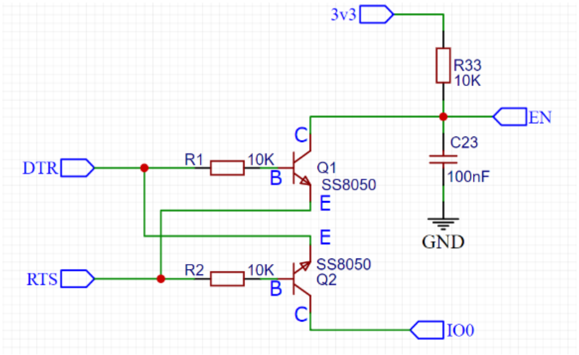


Figure 102 Auto configure circuit with RC delay.

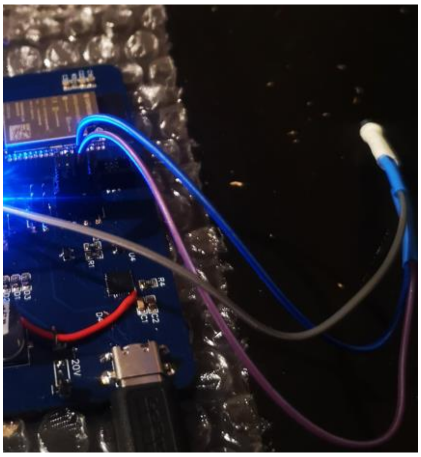


Figure 103 10uf EN FIX blue purple gray wire

During the design phase, the datasheet recommended using a 10k resistor and a 0.1uF capacitor in the RC delay circuit. However, after facing this issue, we conducted further investigation into the power schemas for the module. We discovered that the manufacturer considers the 10k resistor and 0.1uF capacitor as a baseline and suggests the possibility of requiring additional capacitance in the RC delay circuit. As a result, we added an additional 10uF capacitor between the EN line and GND.

This fix immediately resolved the programming issue, allowing us to successfully program the ESP32 with our program. We were able to receive a reply over the USB-C connection, confirming the successful operation of the module.

8.3.5 Test of LAN quadrant

During the design phase, it was noticed that many designs using the LAN module LAN8720 did not incorporate the required inline reflection inhibit resistors on the data lines, that was recommended in the datasheet. We wanted to evaluate the feasibility of omitting these resistors by conducting measurements with and without them. However, before proceeding with these measurements, it was crucial to ensure that the LAN module was functioning properly.

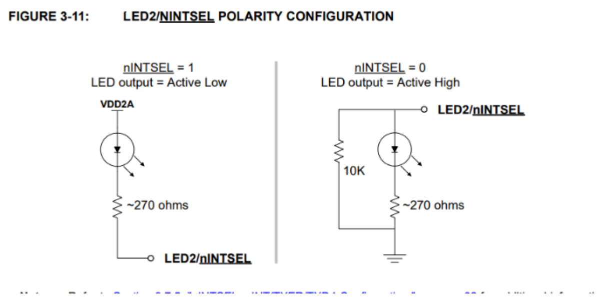
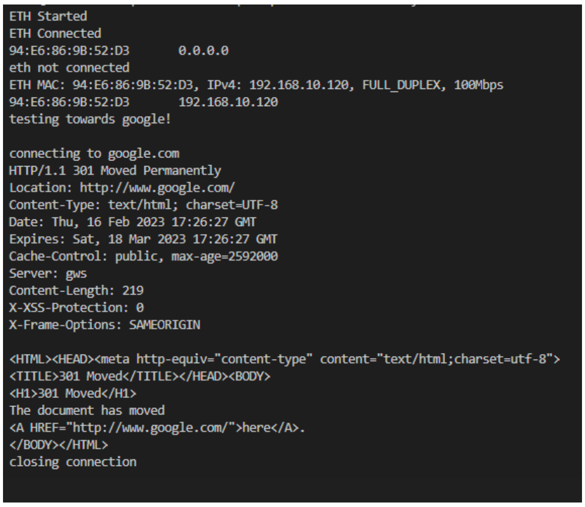


Figure 104 Recommended strapping pin config

Initially, when connecting the LAN8720 quadrant, we attempted to establish communication between the ESP32 and the LAN8720 module by retrieving the module's MAC address using the ESP ETH.H library. However, the LAN8720 module did not respond to any calls from the MCU. Although we confirmed that the module was powered on as we had connected two LEDs to it, this discrepancy prompted us to conduct a thorough investigation of our design.

Upon examination, we discovered that we had inadvertently set one of the strapping pins, nINTSEL, to a low state when it should have been set to high. After reconfiguring the PCB to set nINTSEL = 1, the LAN module powered on successfully, and we obtained the MAC address as intended.



```
ETH Started
ETH Connected
94:E6:86:9B:52:D3       0.0.0.0
eth not connected
ETH MAC: 94:E6:86:9B:52:D3, IPv4: 192.168.10.120, FULL_DUPLEX, 100Mbps
94:E6:86:9B:52:D3       192.168.10.120
testing towards google!

connecting to google.com
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Thu, 16 Feb 2023 17:26:27 GMT
Expires: Sat, 18 Mar 2023 17:26:27 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
closing connection
```

Figure 105 Serial output from ESP32 on internet test

With the MAC address in hand, we proceeded to develop a small program to ping the IP address 8.8.8.8, resulting in our first successful internet connection over an Ethernet cable.

To assess the stability of the Ethernet connection and detect any potential packet losses, we employed the Wireshark application on a computer within the network where the ESP32 was situated. Our testing involved sending a 512-byte ping packet every second for 145 seconds to measure packet loss.

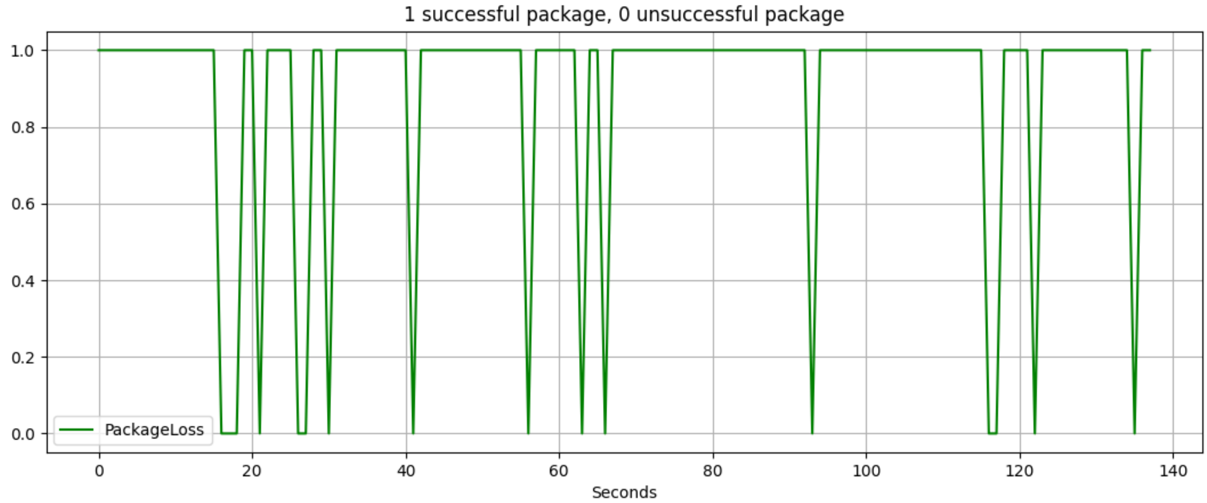


Figure 106 Wireshark network packet analyser data

Unfortunately, these tests revealed a significant issue with our module, as we experienced a high rate of network packet loss.

Considering the extensive investigation we had previously conducted, we were confident that our design adhered to the recommended reference design. Consequently, we proceeded to measure all the lines between the ESP32 and the LAN module. Throughout these measurements, we continuously monitored packet loss between the ESP32 and a computer. It was during the measurement of the CLK line that we observed a sudden, uninterrupted stream of packets. This observation indicated a problem with our CLK line on the PCB.

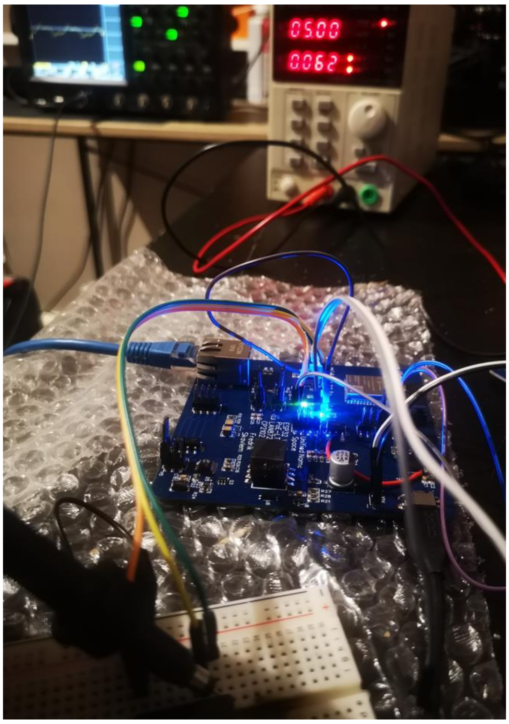


Figure 108 CLK line wired out of PCB for testing

To further analyze and resolve this issue, we introduced a measurement probe to the CLK line, which introduced a 10M ohm resistor and 12pF capacitance between the CLK line and GND, as specified by the probe. This probing helped us identify a potential solution. We began by adding various capacitor values in the range of 12pF to 1nF between the CLK line and GND, attempting to reproduce the effect observed with the probe attached. However, these tests yielded no significant results. Subsequently, we introduced a 1M ohm resistor between the CLK line and GND, resulting in a reduction of packet loss over the network. Further improvement was achieved by adding a 10k resistor between the CLK line and GND, which ultimately eliminated packet loss entirely.

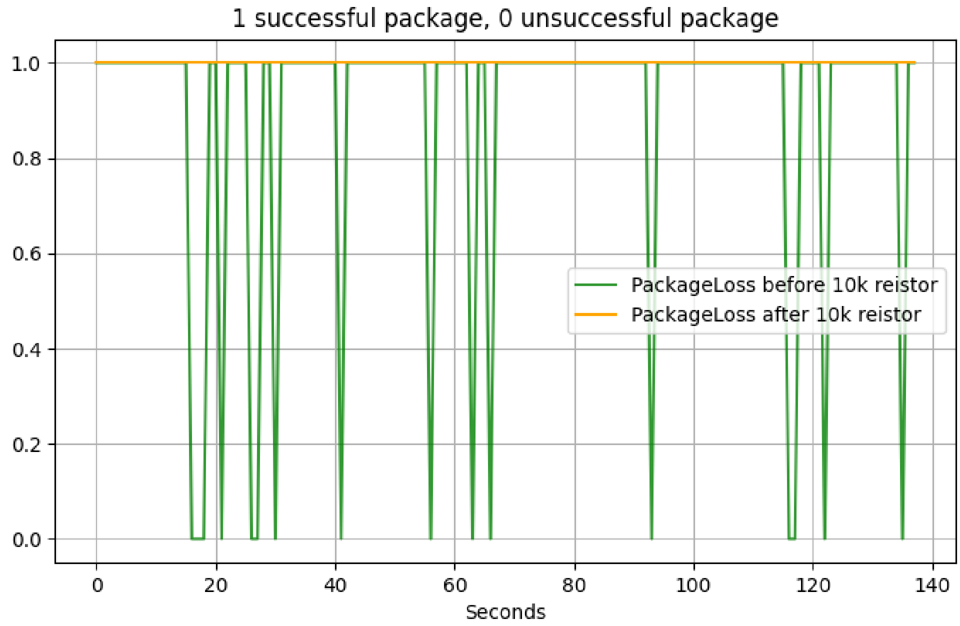


Figure 109 measured packetloss before and after modification

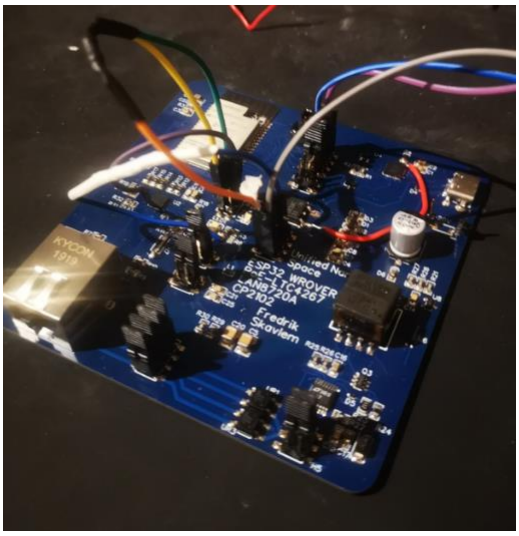


Figure 107 CLK line fix yellow, green, orange wire

Now that the module was operating satisfactorily, we proceeded with testing the reflection resistors. Firstly, we measured the signal of the data lines using an oscilloscope connected to them before and after the modification. We found no signal degradation before or after removing the reflection inhibit resistors. Therefor we replaced them with 0 Ohm resistors, and after the replacement, no packet loss was measured.

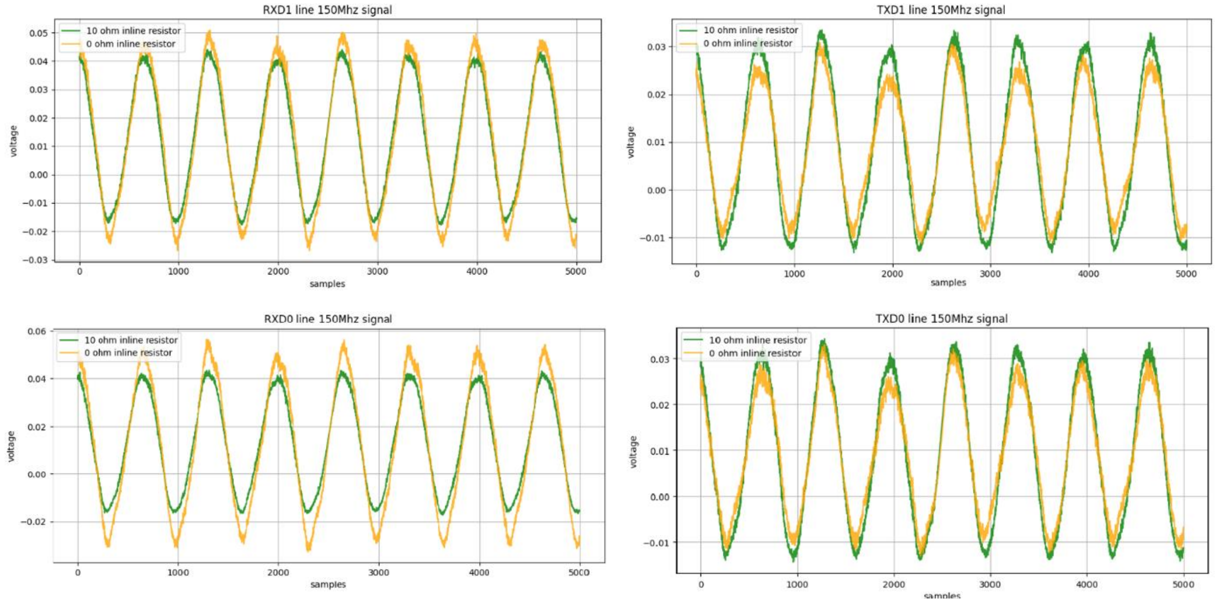
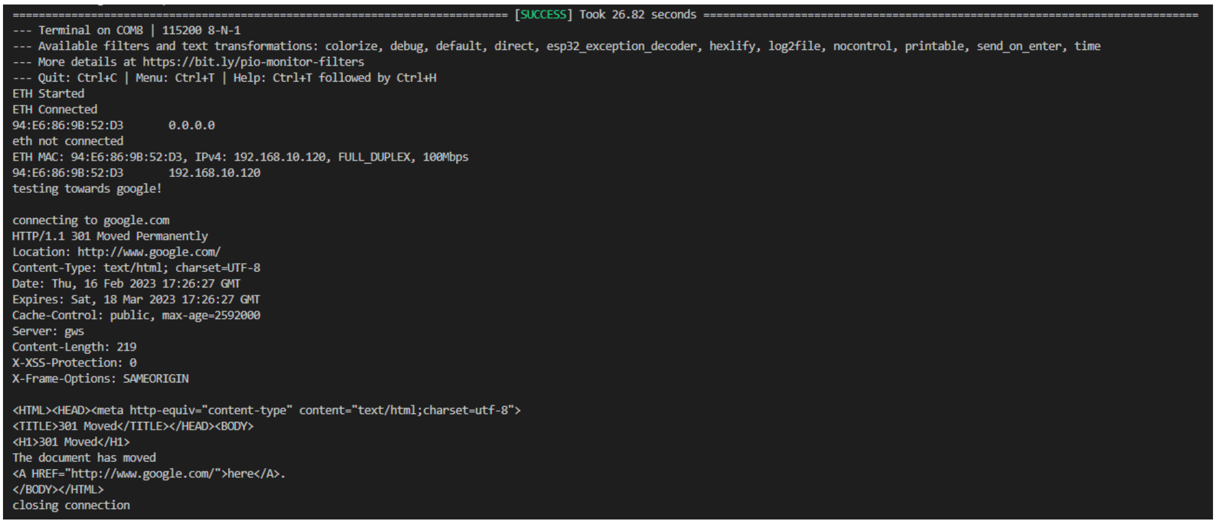


Figure 111 Measurement of datalines with and without reflection resistors



```
=========================================================================== [SUCCESS] Took 26.82 seconds ===========================================================================
--- Terminal on COM8 | 115200 8-N-1
--- Available filters and text transformations: colorize, debug, default, direct, esp32_exception_decoder, hexlify, log2file, nocontrol, printable, send_on_enter, time
--- More details at https://bit.ly/pio-monitor-filters
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
ETH Started
ETH Connected
94:E6:86:9B:52:D3       0.0.0.0
eth not connected
ETH MAC: 94:E6:86:9B:52:D3, IPv4: 192.168.10.120, FULL_DUPLEX, 100Mbps
94:E6:86:9B:52:D3       192.168.10.120
testing towards google!

connecting to google.com
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Thu, 16 Feb 2023 17:26:27 GMT
Expires: Sat, 18 Mar 2023 17:26:27 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
closing connection
```

Figure 110 Connected to internet with IOTT device

8.3.6 Test of Power over ethernet quadrant(PoE)

Upon thorough testing, the group encountered difficulties in getting the PoE quadrant of the PCB to function properly. This quadrant was the last one tested due to the potential risk of damaging the entire PCB. To test the PoE functionality, an active PoE injector was acquired and connected between the WAN outlet and the PCB's Ethernet cable. The injector followed the IEEE 802.3af standard and delivered up to 12.95W of power over Ethernet.

The PoE injector used a pulse to detect PoE consumers, and a handshake was required from the PoE consumer for the injector to supply power above 10mA on the cable. This ensured that non-PoE devices did not receive power over Ethernet. While the group could measure the signature from the injector at the inputs of the LTC4267 PoE module, no handshake response was detected with oscilloscope.

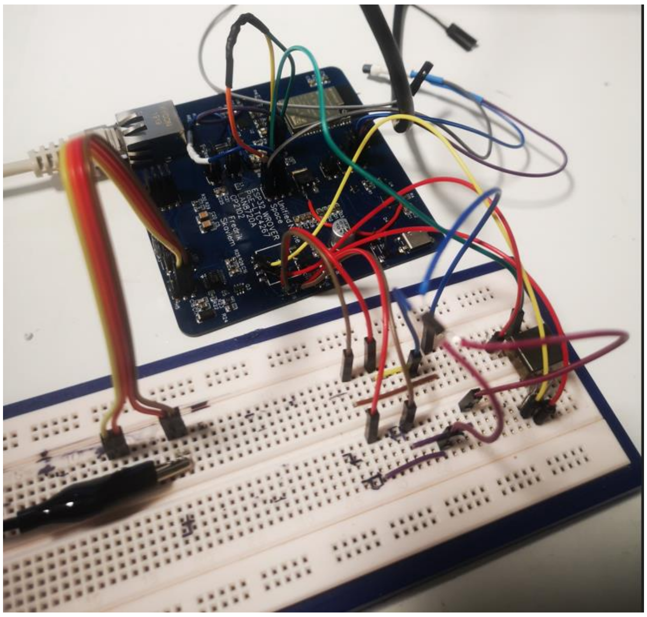
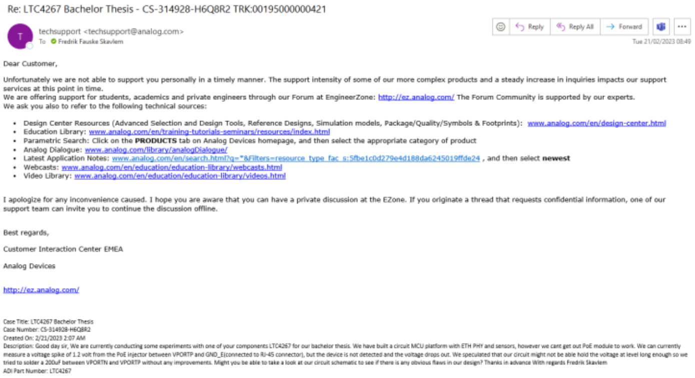


Figure 112 Debugging PoE section

Extensive investigation of the PoE circuit was conducted, comparing it to the recommended circuit, and some flaws were identified. To address these issues, the circuit was modified using a breadboard and DuPont cables for rewiring. Despite these modifications, the handshake still could not be detected. The main issue discovered was that the measured voltage from the PoE injector was only 1.2 volts, whereas the LTC4267 module required a minimum peak voltage of 2.8 volts to respond to the signature.



Re: LTC4267 Bachelor Thesis - CS-314928-H6Q8R2 TRK:00195000000421

techsupport <techsupport@analog.com>
To Fredrik Fauske Skavlem

Tue 21/02/2023 08:49

Dear Customer,

Unfortunately we are not able to support you personally in a timely manner. The support intensity of some of our more complex products and a steady increase in inquiries impacts our support services at this point in time.
We are offering support for students, academics and private engineers through our Forum at EngineerZone: http://ez.analog.com/ The Forum Community is supported by our experts.
We ask you also to refer to the following technical sources:

- Design Center Resources (Advanced Selection and Design Tools, Reference Designs, Simulation models, Package/Quality/Symbols & Footprints): www.analog.com/en/design-center.html
- Education Library: www.analog.com/en/training-tutorials-seminars/resources/index.html
- Parametric Search: Click on the **PRODUCTS** tab on Analog Devices homepage, and then select the appropriate category of product
- Analog Dialogue: www.analog.com/library/analogDialogue/
- Latest Application Notes: www.analog.com/en/search.html?q=*&Filters=resource_type_fac_s:5fbe1c0d279e4d188da6245019ffde24 , and then select **newest**
- Webcasts: www.analog.com/en/education/education-library/webcasts.html
- Video Library: www.analog.com/en/education/education-library/videos.html

I apologize for any inconvenience caused. I hope you are aware that you can have a private discussion at the EZone. If you originate a thread that requests confidential information, one of our support team can invite you to continue the discussion offline.

Best regards,

Customer Interaction Center EMEA

Analog Devices

http://ez.analog.com/

Case Title: LTC4267 Bachelor Thesis
Case Number: CS-314928-H6Q8R2
Created On: 2/21/2023 2:07 AM
Description: Good day sir, We are currently conducting some experiments with one of your components LTC4267 for our bachelor thesis. We have built a circuit MCU platform with ETH PHY and sensors, however we cant get out PoE module to work. We can currently measure a voltage spike of 1.2 volt from the PoE injector between VPORTP and GND_E(connected to RJ-45 connector), but the device is not detected and the voltage drops out. We speculated that our circuit might not be able hold the voltage at level long enough so we tried to solder a 200uF between VPORTN and VPORTP without any improvements. Might you be able to take a look at our circuit schematic to see if there is any obvious flaws in our design? Thanks in advance With regards Fredrik Skavlem
ADI Part Number: LTC4267

Figure 113 Reply from analog devices

Further investigation revealed that Analog Devices, the supplier of the LTC4267 module, did not use the recommended circuit from the datasheet in any of their reference designs involving the module. The group reached out to Analog Devices for support, but unfortunately, the company was unable to investigate the issue. They suggested posting the problem on their engineering zone forums, though no solutions have been found so far.

Considering the challenges faced and the fact that the PoE module was always intended as an attachable component, the group made the decision not to implement the attachable board for PoE on the final design. However, they kept the pin headers open on the final design, allowing for the possibility of adding a PoE device directly to the PCB as a module in future experimentation.

8.3.7 Overview of reworks done

After rigorously testing our PCB a total of 9 reworks was done, including the failure of the entire PoE section. These reworks will be included in the final design of our PCB microcontroller.

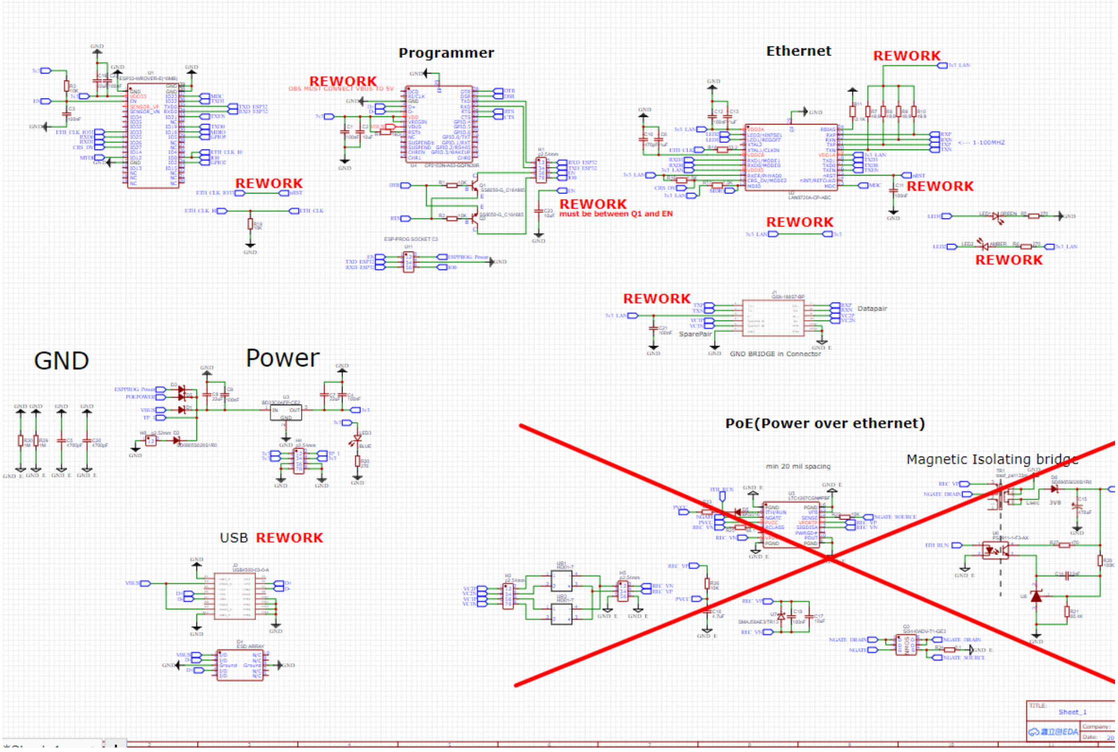


Figure 114 IIOT Schematic, all reworks marked in red

8.4 Final design

Final design worked satisfactory, we added some additional components, a sensor and a microSD card holder to increase complexity. The sensor was a barometric sensor and used I2C protocol. This was tested and found satisfactory. The entire POE section was redesigned as mentioned in chapter 4.3, however due to time constraints in this project, the new POE section never got tested. Since the PoE section was a module to the main PCB it was not necessary for the main PCB to operate.

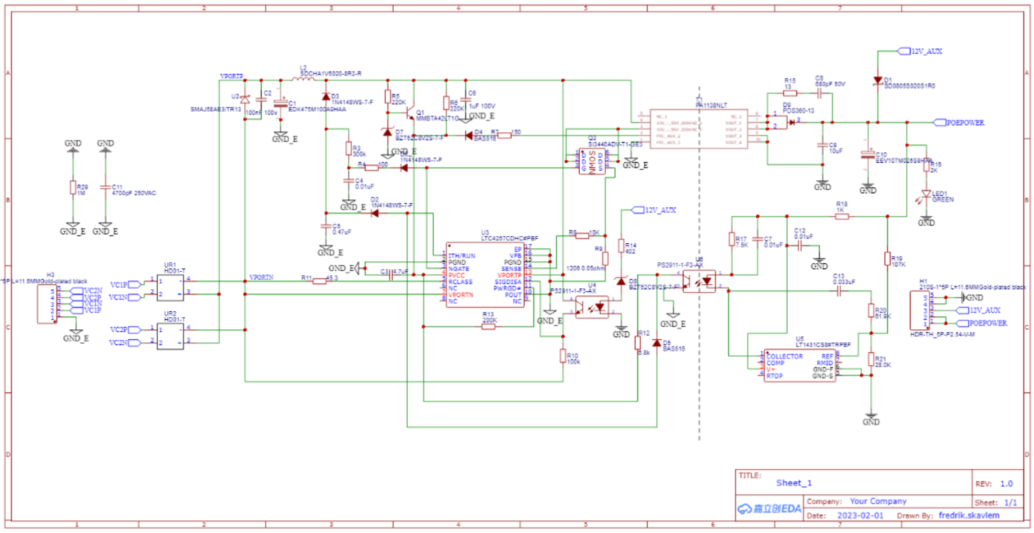
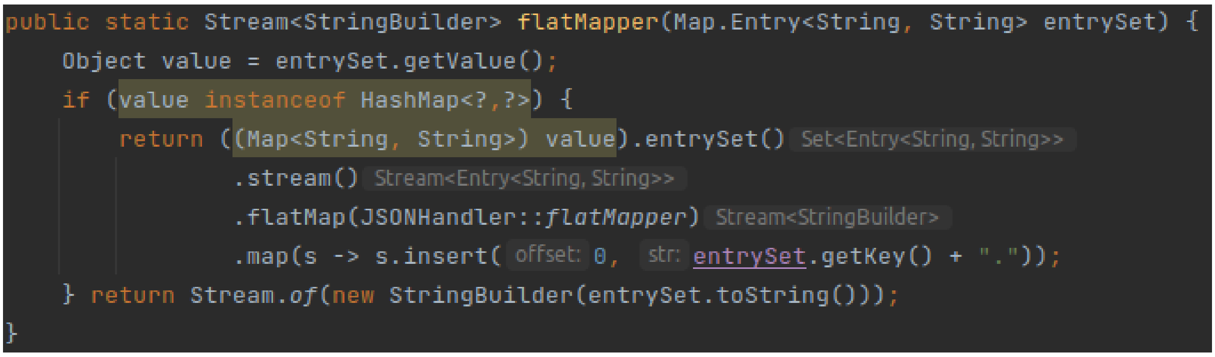


Figure 115 Final PoE design. Not manufactured

9 APPENDIX D – Historian JSON parsing code from 4.4

FlattenHashMap starts off with taking a HashMap and sending it off to FlatMapper.

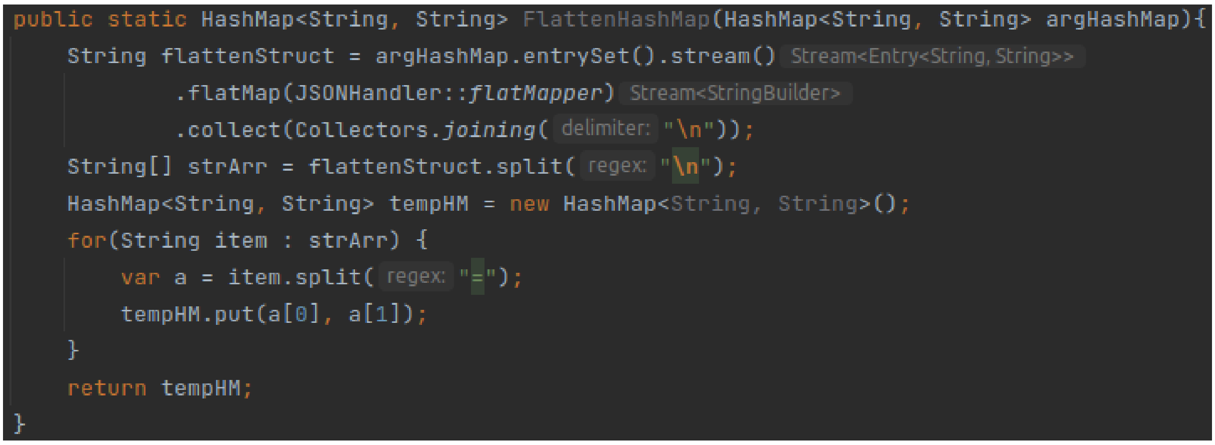
FlatMapper takes a Map.Entry object as input and retrieves the value of the current set. If the value within the key-value set is a HashMap, the method recursively calls itself and appends the key of the list to each value. After it has done this for every single nested map, it returns the new HashMap as a Stream of Stringbuilder objects, containing the entrySet as a String, one set at a time.



```
public static Stream<StringBuilder> flatMapper(Map.Entry<String, String> entrySet) {
    Object value = entrySet.getValue();
    if (value instanceof HashMap<?,?>) {
        return ((Map<String, String>) value).entrySet() Set<Entry<String, String>>
                .stream() Stream<Entry<String, String>>
                .flatMap(JSONHandler::flatMapper) Stream<StringBuilder>
                .map(s -> s.insert( offset: 0,  str: entrySet.getKey() + "."));
    } return Stream.of(new StringBuilder(entrySet.toString()));
}
```

Figure 116 FlatMapper method for finding the innermost HashMap in the JSON string

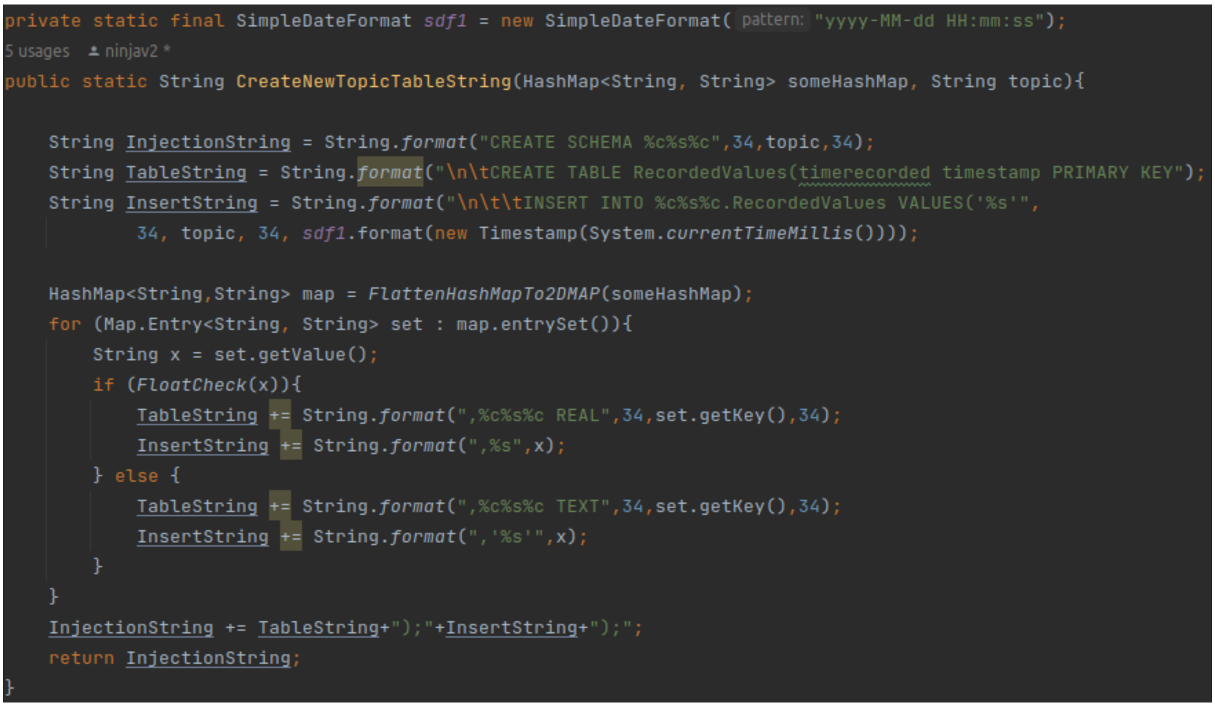
FlattenHashMap then takes the sets it receives and flattens them into a single string, which is delimited by a newline character. This string then gets put into an array based on these newline characters. We iterate through the array, splitting each item at the equal sign and putting the key-value pairs into the temporary HashMap that the method creates. Then finally we return the temporary HashMap.



```
public static HashMap<String, String> FlattenHashMap(HashMap<String, String> argHashMap){
    String flattenStruct = argHashMap.entrySet().stream() Stream<Entry<String, String>>
            .flatMap(JSONHandler::flatMapper) Stream<StringBuilder>
            .collect(Collectors.joining( delimiter: "\n"));
    String[] strArr = flattenStruct.split( regex: "\n");
    HashMap<String, String> tempHM = new HashMap<String, String>();
    for(String item : strArr) {
        var a = item.split( regex: "=");
        tempHM.put(a[0], a[1]);
    }
    return tempHM;
}
```

Figure 117 The FlattenHashMap method

CreateNewTopicTableString is one of two methods that is responsible for taking the HashMap's and creating SQL queries for inserting them into our database. Firstly, it initializes three different strings which are called InjectionString, TableString and InsertString. InjectionString creates a new schema with the correct topic ID. TableString creates a table within the schema, calls it RecordedValues and sets the timestamp for when this information was received as a primary key. Lastly, the InsertString gets all the values from the HashMap. After this is done, we have a full SQL query that holds our latest payload which is now ready for storage. This query gets sent off to R2DBC.



```java
private static final SimpleDateFormat sdf1 = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss");
5 usages  ninjav2 *
public static String CreateNewTopicTableString(HashMap<String, String> someHashMap, String topic){

    String InjectionString = String.format("CREATE SCHEMA %c%s%c",34,topic,34);
    String TableString = String.format("\n\tCREATE TABLE RecordedValues(timerecorded timestamp PRIMARY KEY");
    String InsertString = String.format("\n\t\tINSERT INTO %c%s%c.RecordedValues VALUES('%s'",
            34, topic, 34, sdf1.format(new Timestamp(System.currentTimeMillis())));

    HashMap<String,String> map = FlattenHashMapTo2DMAP(someHashMap);
    for (Map.Entry<String, String> set : map.entrySet()){
        String x = set.getValue();
        if (FloatCheck(x)){
            TableString += String.format(",%c%s%c REAL",34,set.getKey(),34);
            InsertString += String.format(",%s",x);
        } else {
            TableString += String.format(",%c%s%c TEXT",34,set.getKey(),34);
            InsertString += String.format(",'%s'",x);
        }
    }
    InjectionString += TableString+");"+InsertString+");";
    return InjectionString;
}
```

Figure 118 CreateNewTopicTableString method responsible for creating SQL strings.

Now that all values and their names from the JSON string were de-nested and put into a HashMap in a timely manner, we could move on to inserting this information into the database. The next two methods that were written take the HashMap passed to them and create SQL queries for inserting this information into our database. The difference between them is that “CreateNewTopicTableString” creates a new schema for the table which holds the information. While “InsertTopicTableString” inserts the data into an already existing schema and table. In short, “CreateNewTopicTableString” is used when the topic and message arrive for the first time, while the other method is used if it’s a topic that already has been received before. In addition, every parameter passed into the SQL query is checked by FloatCheck, to see if it’s a number than can be stored as a float or not.

10 APPENDIX E - PLC OPC server configuration

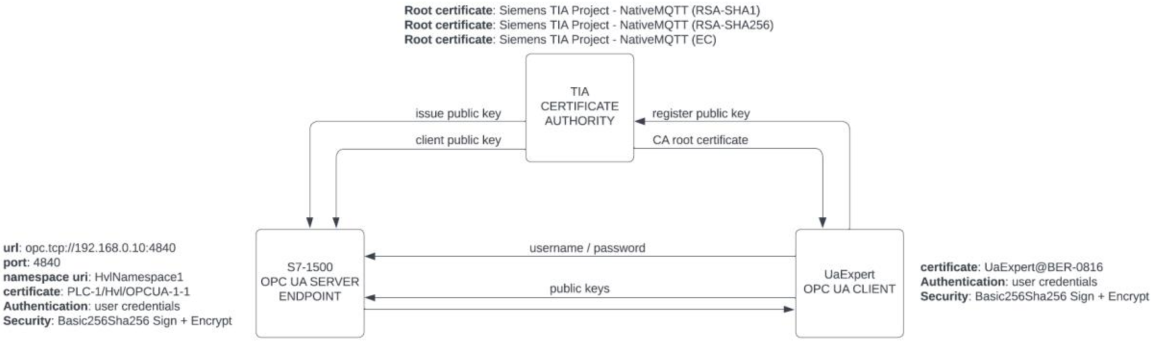
The PLC logic and the OPC server have been programmed using TIA portal, a software package provided by Siemens for the development of automation systems.

10.1 Setting up CA management in TIA portal

TIA portal can function as a certification authority, creating certificates for subjects needing them for authentication and security. The advantage of using certificates signed by TIA portal is that other clients must import only the CA certificate. Then any number of certificates can be created for any number of devices, automatically trusted by anyone that has imported the TIA certificate. Two types of certificates can be created.

- **Self-signed certificates** must be imported or trusted by the communicating party before secure communication can be established. Used cases are primarily for topologies with very few devices.

- **Certificates signed by CA** (TIA portal). Only the CA certificate must be imported by the communicating party before secure communication can be established.



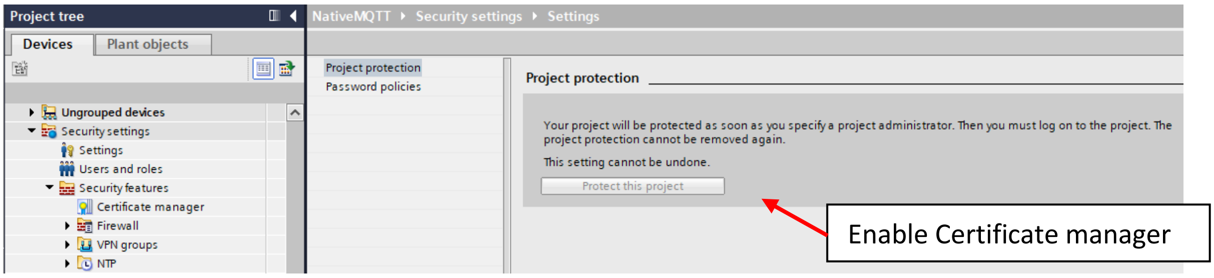
Root certificate: Siemens TIA Project - NativeMQTT (RSA-SHA1)
Root certificate: Siemens TIA Project - NativeMQTT (RSA-SHA256)
Root certificate: Siemens TIA Project - NativeMQTT (EC)

url: opc.tcp://192.168.0.10:4840
port: 4840
namespace uri: HvlNamespace1
certificate: PLC-1/Hvl/OPCUA-1-1
Authentication: user credentials
Security: Basic256Sha256 Sign + Encrypt

certificate: UaExpert@BER-0816
Authentication: user credentials
Security: Basic256Sha256 Sign + Encrypt

Figure 119 Configuration setup of TIA portal as CA and S7-1500 as an OPC UA Server

After project creation, the first step is to enable “Project protection” to enable CA management.



Your project will be protected as soon as you specify a project administrator. Then you must log on to the project. The project protection cannot be removed again.

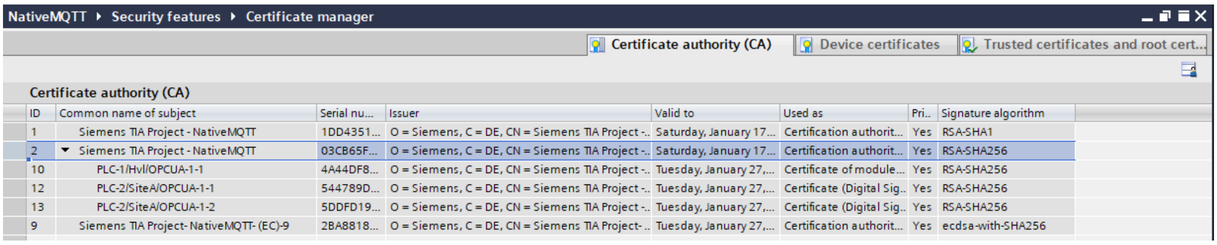
This setting cannot be undone.

Enable Certificate manager

Figure 120 Enabling the certificate store in TIA portal

Then TIA portal will create a certificate store and three root certificates that will be used to sign and issue other certificates. The root certificates have different levels of security that can be chosen dependent on the desired protection, the effort needed to encrypt and sign packets, or backward compatibility for devices that have not yet implemented the most recent version of the security protocols.

- **RSA-SHA1** is due to be replaced by newer hashing algorithms.
- **RSA-SHA256** is the most common security setting in use today.
- **ECDSA-with-SHA256** has improved security at an additional computational cost. Elliptic curve cryptography is a relatively new technology and has not yet been implemented by all suppliers.



NativeMQTT ▸ Security features ▸ Certificate manager

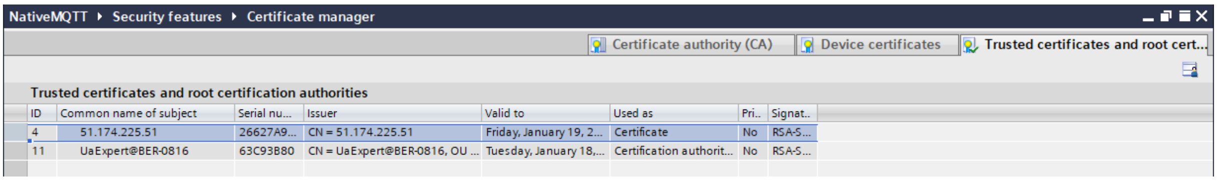
Certificate authority (CA) | Device certificates | Trusted certificates and root cert...

Certificate authority (CA)

ID	Common name of subject	Serial nu...	Issuer	Valid to	Used as	Pri..	Signature algorithm
1	Siemens TIA Project - NativeMQTT	1DD4351...	O = Siemens, C = DE, CN = Siemens TIA Project -..	Saturday, January 17...	Certification authorit...	Yes	RSA-SHA1
2	▾ Siemens TIA Project - NativeMQTT	03CB65F...	O = Siemens, C = DE, CN = Siemens TIA Project -..	Saturday, January 17...	Certification authorit...	Yes	RSA-SHA256
10	PLC-1/Hvl/OPCUA-1-1	4A44DF8...	O = Siemens, C = DE, CN = Siemens TIA Project -..	Tuesday, January 27,...	Certificate of module...	Yes	RSA-SHA256
12	PLC-2/SiteA/OPCUA-1-1	544789D...	O = Siemens, C = DE, CN = Siemens TIA Project -..	Tuesday, January 27,...	Certificate (Digital Sig..	Yes	RSA-SHA256
13	PLC-2/SiteA/OPCUA-1-2	5DDFD19...	O = Siemens, C = DE, CN = Siemens TIA Project -..	Tuesday, January 27,...	Certificate (Digital Sig..	Yes	RSA-SHA256
9	Siemens TIA Project- NativeMQTT- (EC)-9	2BA8818...	O = Siemens, C = DE, CN = Siemens TIA Project- ..	Tuesday, January 27,...	Certification authorit...	Yes	ecdsa-with-SHA256

Figure 121 TIA certificates and client certificates signed by TIA root certificate

Self-signed certificates from devices or external communication parties can be imported and managed in the TIA portal. These certificates can then be assigned to devices or PLCs' trust lists to enable authentication and secure communication.



NativeMQTT ▸ Security features ▸ Certificate manager

Certificate authority (CA) | Device certificates | Trusted certificates and root cert...

Trusted certificates and root certification authorities

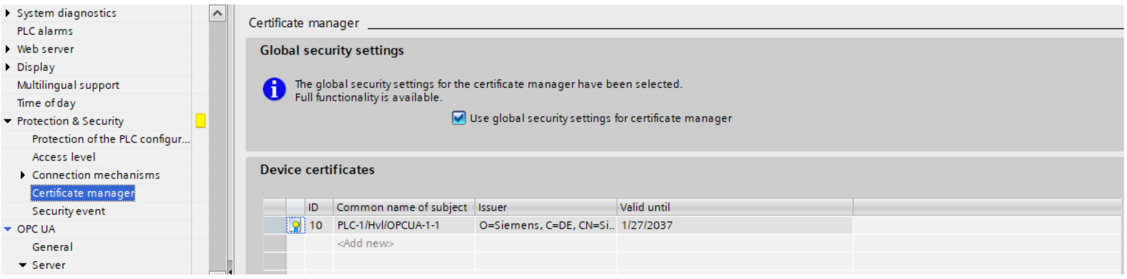
ID	Common name of subject	Serial nu...	Issuer	Valid to	Used as	Pri..	Signat..
4	51.174.225.51	26627A9...	CN = 51.174.225.51	Friday, January 19, 2...	Certificate	No	RSA-S...
11	UaExpert@BER-0816	63C93B80	CN = UaExpert@BER-0816, OU ...	Tuesday, January 18,...	Certification authorit...	No	RSA-S...

Figure 122 Imported certificates added to PLCs' trust lists.

Timing is an important attribute in secure communication. Most cyber security protocols include timestamps inside the messages to prevent replay attacks. Therefore, ensuring that the internal clocks on the devices are synchronized is important. A robust solution commonly implemented is to set up a Network Time Protocol (NTP) server that all other devices poll to synchronize. The alternative is to manually set the system clock. The disadvantage is that the clock might deviate over time, suddenly making the communication fail because of timestamp differences. An encrypted packet will be discarded if the timestamp exceeds the defined timestamp limit.

10.2 Enabling of OPC UA server

Each device needs to have a certificate assigned to it. In TIA portal it is possible to link the certificate manager to the project PLCs inside the device configuration. Then a certificate can be assigned to the device. It will be the same certificate that communicating parties need to import and trust.



Certificate manager

Global security settings

The global security settings for the certificate manager have been selected.
Full functionality is available.

☑ Use global security settings for certificate manager

Device certificates

ID	Common name of subject	Issuer	Valid until
10	PLC-1/Hvl/OPCUA-1-1	O=Siemens, C=DE, CN=Si..	1/27/2037
	<Add new>		

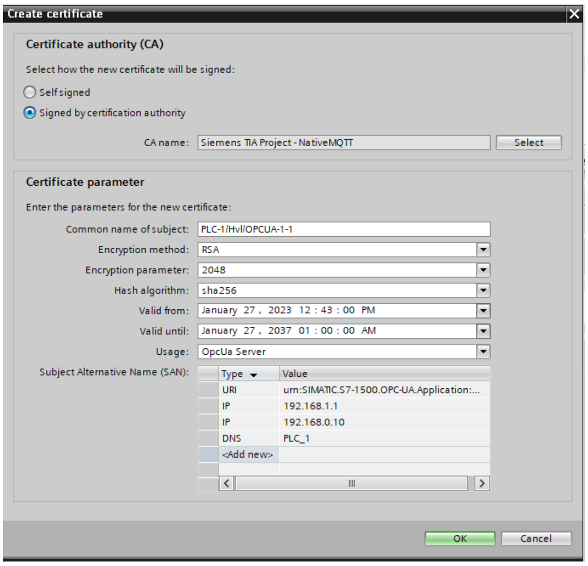
Figure 123 Certificate manager enabled for project devices

A check box inside the device configuration enables the server. A license is needed for the server to function and needs to be assigned during configuration. The most important parameters of the server are:

Server Addresses: url of the server endpoint: opc.tcp://192.168.1.10:4840
Namespace uri: used to organize variables in groups: WaterControllerInterface
Port: application port: 4840
Min/Max parameters: limits number of sessions, timeout, request intervals, etc.

10.3 Configuring Authentication and Security

The server's certificate needs to be created with the necessary subject details. The subject name is the most critical attribute that needs to include the URI, IP addresses, and possibly the DNS name that translates to the same IP address. The name will be verified by the client and rejected if coincidental with the communicating party. The other attributes affect the security level of the key and can be selected according to requirements. The certificate is signed by the root CA (TIA) so that we can avoid importing it to every client communicating with the server.



Certificate authority (CA)

Select how the new certificate will be signed:

○ Self signed
● Signed by certification authority

CA name: Siemens TIA Project - NativeMQTT

Certificate parameter

Enter the parameters for the new certificate:

Common name of subject: PLC-1/Hvl/OPCUA-1-1
Encryption method: RSA
Encryption parameter: 2048
Hash algorithm: sha256
Valid from: January 27 , 2023 12 : 43 : 00 PM
Valid until: January 27 , 2037 01 : 00 : 00 AM
Usage: OpcUa Server
Subject Alternative Name (SAN):

Type	Value
URI	urn:SIMATIC.S7-1500.OPC-UA.Application:...
IP	192.168.1.1
IP	192.168.0.10
DNS	PLC_1
<Add new>	

Figure 124 S7-1500 server certificate

The main security settings are the type of authentication and security during the communication session. Most OPC UA devices support Basic256Sha256 – Sign and Encrypt. It should therefore be set as the default security level during configuration. A lower level of security should only be chosen if the communication party is not supporting the default. The configuration of the server has all other options turned off.

Authentication is primarily as anonymous, with username and password or by certificates. Siemens S7-1500 does not support certificate authentication and is therefore configured with a password. The security during session communication is signed with the client and server certificates and will not function without the certificate being trusted by both parties. In that way, it is also authentication using certificates, even though it is not performed during session authentication.

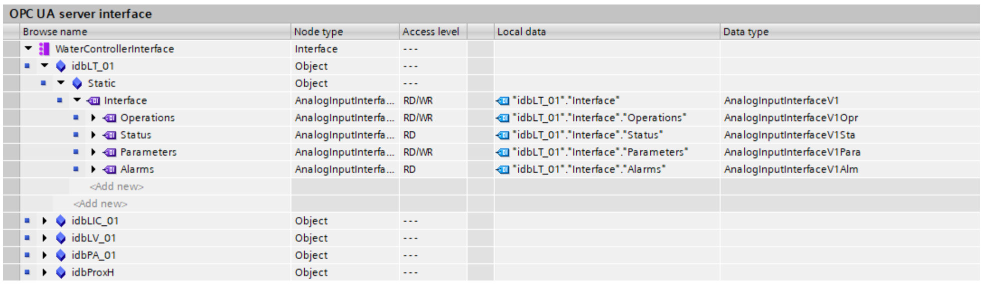
10.4 OPC UA gateway client configuration

OPC UA gateway is a client developed for mapping data between the OPC and MQTT interfaces. The certificate of the client has already been imported into the server. To enable secure communication with the server, we must also import the PLC server certificate into the client, which can be done through a dialog during session establishment. The client will verify the subject name against a DNS server. Since we have not registered our PLC with a DNS, it is possible to add PLC_2's IP address to the windows hosts file. It will still work without registering, but then we will have to click past a security warning every time we connect the client to the server. Another possibility is to export the TIA root certificate and add it to windows certificate store, thereby authenticating the PLC during session establishment.

10.5 Structuring data

The OPC UA specification provides a valuable prerequisite with the browse service, which enables the integrators to connect to the server to browse its address space and then determine the correct node address for the variable. Those who will connect to the server to retrieve data are most likely not the same ones who configured it. Adding context to the data is essential to make this job more manageable. Browsing has been made intuitive by comments and organizing data in the object-oriented and hierarchical structure inspired by how the physical process is laid out. An advantage of this is that it can easily be extended outside our small example process by adding an extra prefix to the symbol names when publishing on topics so that the hierarchy becomes a little deeper.

Information on OPC UA servers is organized in namespaces. Namespaces 0 and 1 are predefined by the OPC Foundation for storing data types and metadata needed for handling the data. Every OPC server needs to implement these. The process data has been organized on namespace 4, named "WaterControllerInterface". It only contains relevant data to the process.



OPC UA server interface

Browse name	Node type	Access level	Local data	Data type
WaterControllerInterface	Interface	---		
idbLT_01	Object	---		
Static	Object	---		
Interface	AnalogInputInterfa...	RD/WR	"idbLT_01"."Interface"	AnalogInputInterfaceV1
Operations	AnalogInputInterfa...	RD/WR	"idbLT_01"."Interface"."Operations"	AnalogInputInterfaceV1Opr
Status	AnalogInputInterfa...	RD	"idbLT_01"."Interface"."Status"	AnalogInputInterfaceV1Sta
Parameters	AnalogInputInterfa...	RD/WR	"idbLT_01"."Interface"."Parameters"	AnalogInputInterfaceV1Para
Alarms	AnalogInputInterfa...	RD	"idbLT_01"."Interface"."Alarms"	AnalogInputInterfaceV1Alm
<Add new>				
<Add new>				
idbLIC_01	Object	---		
idbLV_01	Object	---		
idbPA_01	Object	---		
idbProxH	Object	---		

Figure 125 S7-1500 Upc Ua server interface

Manipulating variables on the OPC server should not be possible for anyone. A false and possibly unrealistic view of the process can occur if changed uncritically. Changing commands or setpoints can be even more devastating. Separate Read, Write, or Read/Write access can be configured individually for each variable and is part of the authorization system needed to prevent unauthorized manipulation of the variables and a potentially dangerous process state. All statuses are set to read-only. More fine-grained access control for commands is configured in the MQTT data broker, restricting access to topics for unauthorized clients.

11 APPENDIX F - OPC to MQTT gateway

11.1 Conceptual design

The gateway is created in C# using .NET 6.0 standard, which has many built-in libraries and distributed Nuget packages for common services. Considerable emphasis has been put on the user experience and UI. A well-designed user interface is essential to make the application intuitive and to reduce the engineer's integration time and cost. Therefore, it was decided to create it as a Windows Presentation Forms (WPF) application which has a particularly nice-looking User Interface and scales well to different resolutions and window sizes. The application also has the function to be minimized to the taskbar. It can run in the background after the configuration of data exchange between the OPC and MQTT interfaces has been configured. On the taskbar, it will not be distracting to the user and requires fewer resources from the operating system because a graphical user interface does not need to be rendered. The application is set to launch in the taskbar upon start-up, and the option for it to be included in the list of windows startup applications can be enabled in the configuration menu within the application.

The Siemens OPC client tutorial[50] has been used to learn how sessions work and as inspiration during the application development. The tutorial is using NetStandard.Opc.UA distributed by the OPC Foundation, a library available as a NuGet package for handling services defined in the specification. It will be added to the project in visual studio. It includes classes for handling communication between the client in the gateway and the OPC server that resides in the PLC. The library is under General Public License (GPL) 2.0 license making it available to anybody, however, it requires the user to disclose the application code if the user is not a member of the OPC Foundation. Even when using a distributed library, it requires detailed knowledge of how services are used inside OPC sessions and how data is structured on a server address space to be able to retrieve it efficiently. Online debugging in Visual Studio has been an invaluable tool for understanding results retrieved from the functions defined in the NetStandard.Opc.UA library. Knowledge about the structure of cryptographic certificates and how certificate stores are managed has also been essential since communication security relies heavily on these technologies.

The idea behind Appendix F is to describe the overall structure, how the gateway protocol mapper is intended to work, and what design choices have been made. Small extracts of the code have been included when it makes sense to gain a better understanding of the gateway structure. It is not sufficient to or the intent that the reader has a complete understanding of the entire application, which is attached as an appendix to the thesis.

The words **items**, **nodes** and **variables** are often used interchangeably. To summarize, data is structured as nodes in the OPC server. A node is either a variable with attached metadata or a kind of bucket without a value but with references to organize the address space. An item in the OPC specification is often used to describe variables configured in a monitoring list. The word item is further used as an expression for variables that are configured/mapped between the two interfaces.

A description of common design technics and principles like Dependency Injection can be found at the end of Appendix F.

11.2 Overall application structure

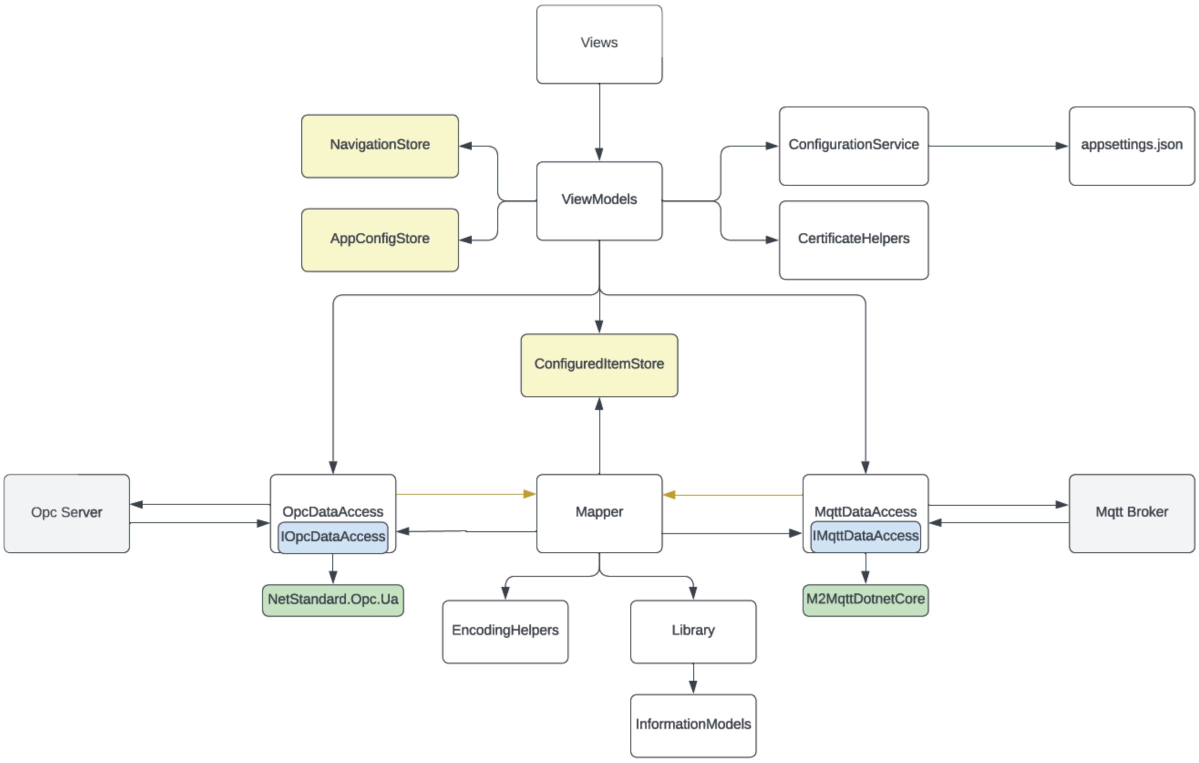


Figure 126 Overview of OPC Gateway classes and relationships

Application functionality has been organized into different classes based on similarity, making it easier to expand, replace individual modules, or move the entire application to a different user interface.

Views and **ViewModels** contain the properties and methods needed to interact with the user. These classes will call on services and library classes that handle communications and translator logic.

Business logic is implemented in UI-independent service classes like DataAccess and is instantiated and added to dependency injection at application startup. These classes implement logic to handle specific functions like communication to the OPC server endpoint. They are designed to be easily interchangeable with other classes for the same functionality or to move the application to a different user interface.

Helper classes and services implement methods for basic functionality like reading to and from the application configuration, encoding messages before transmission, or checking certificates. The caller class provides the required parameters and fetches the result of these operations with no considerations for internal processing within the helper class.

Stores are used for variables that several classes in the app domain need to access. Using stores guarantees consistency and makes adding, updating, or deleting global application data more robust by enforcing restrictions and implementing events to notify subscribers when data changes.

Information models are separated into a class library to maintain independence from the gateway project and to enable easy distribution to others.

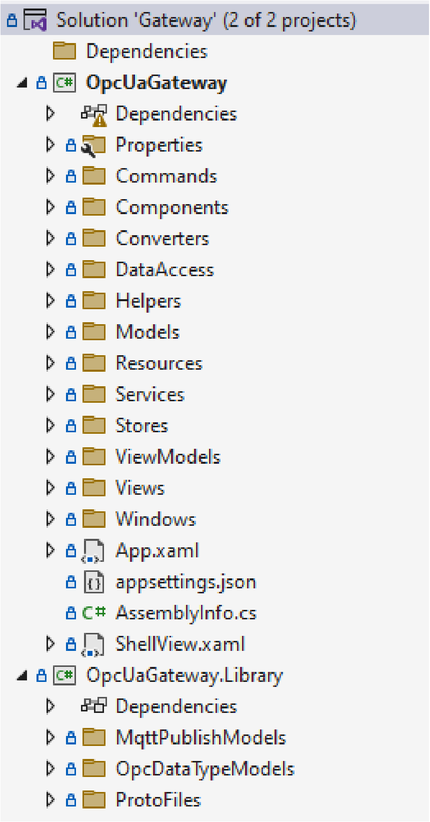
External libraries like NetStandard.Opc.Ua and M2MqttDotnetCore are included in the project to handle complex logic associated with the OPC and MQTT standards.

11.3 Project organization

The Gateway application is divided into two projects in Visual Studio. The most comprehensive is a WPF project containing all the classes that make up the graphical interface and data access to the two endpoints. The second project is a NetStandard class library that stores the information models.

The code is modularized in classes to make it easier to maintain and simplifies changes without redesigning the entire framework. The WPF project is further divided into 12 folders to structure the various classes according to their function.

1. **Commands**: Commands linked to buttons on the UI with a global scope
2. **Components**: Graphical user controls, one for each information model
3. **Converters**: Transform Boolean/Int variables to UI visibility properties
4. **DataAccess**: Handles communication interfaces
5. **Helpers**: Static methods to execute subfunctions
6. **Models**: Organize data in structures internal to the application
7. **Resources**: Graphics used on the UI
8. **Services**: Mapping data and read/write to appsettings, included in DI
9. **Stores**: Provides access to global variables through DI
10. **ViewModels**: Code behind the different Views
11. **Views**: Graphical user interfaces separated into pages
12. **Windows**: Dialogs or UI components used inside pages



```
Solution 'Gateway' (2 of 2 projects)
    Dependencies
  OpcUaGateway
      Dependencies
      Properties
      Commands
      Components
      Converters
      DataAccess
      Helpers
      Models
      Resources
      Services
      Stores
      ViewModels
      Views
      Windows
      App.xaml
      appsettings.json
      AssemblyInfo.cs
      ShellView.xaml
  OpcUaGateway.Library
      Dependencies
      MqttPublishModels
      OpcDataTypeModels
      ProtoFiles
```

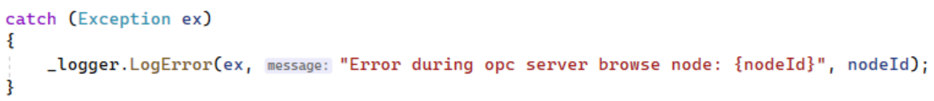
Figure 127 OPC Gateway project organization

Separating the information models into a different library forces loose couplings to the GUI project since references to components inside the WPF application or use of dependencies injection is impossible. The library consists essentially of models used to structure data.

1. **MqttPublishModels**: Defines data formats of information published to the MQTT cluster
2. **OpcDataTypeModels**: OPC data type definitions to guarantee compatible commands from MQTT
3. **ProtoFiles**: Protobuf contracts for distribution but not used in the project

11.4 Logger

Logging can provide valuable information during development or after the application is released. A good log will provide a chronological list of configured log events, including timestamps when they occurred. These events can provide valuable information for identifying and correcting errors that are not discovered until the application is in production. Serilog, a well-developed and maintained Nuget library, has been included in the gateway application to create a log file. Any event, including additional information, can be passed to the Serilog logger based on application triggers, etc. try-catch statements.

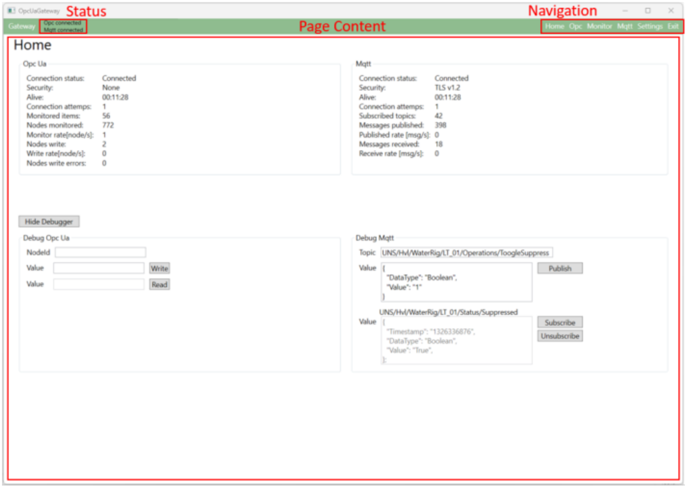


```
catch (Exception ex)
{
    _logger.LogError(ex, message: "Error during opc server browse node: {nodeId}", nodeId);
}
```

Figure 128 Example of logging exceptions with additional information using Serilog.

11.5 Graphical user interface

11.5.1 Shell view



Home

Opc Ua

Connection status: Connected
Security: None
Alive: 00:11:28
Connection attemps: 1
Monitored items: 56
Nodes monitored: 772
Monitor rate[node/s]: 1
Nodes write: 2
Write rate[node/s]: 0
Nodes write errors: 0

Mqtt

Connection status: Connected
Security: TLS v1.2
Alive: 00:11:28
Connection attemps: 1
Subscribed topics: 42
Messages published: 398
Published rate [msg/s]: 0
Messages received: 18
Receive rate [msg/s]: 0

Debug Opc Ua

NodeId
Value
Value

Debug Mqtt

Topic UNS/Hvl/WaterRig/LT_01/Operations/ToogleSuppress

Value
```
{
  "DataType": "Boolean",
  "Value": "1"
}
```

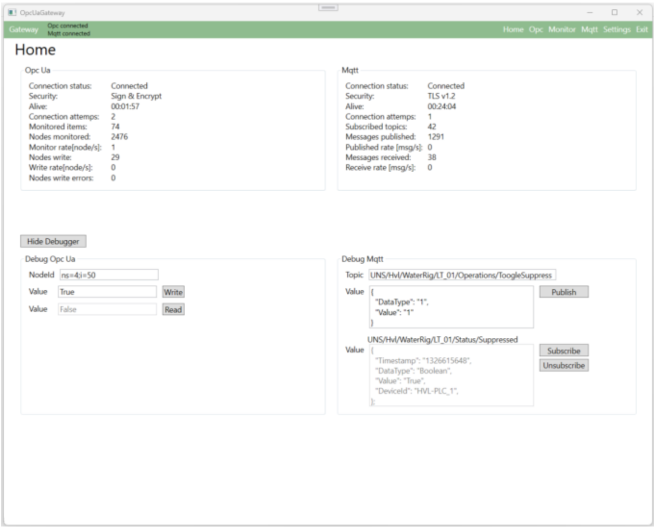
UNS/Hvl/WaterRig/LT_01/Status/Suppressed

Value
```
{
  "Timestamp": "1326336876",
  "DataType": "Boolean",
  "Value": "True",
};
```

Figure 129 Gateway application shell view

The graphical user interface is designed with a frame common to all the different views. Common content is located on this shell, having a separate View-ViewModel to avoid recreating it on all the pages. Navigation is located in the top right corner enabling to switch between most of the pages in the application, except for some settings menus. In the top left is the connection status of the two interfaces shown. The text will change to red if the connection to one of the servers has failed. The lower part of the UI contains the page content for the various Views and will vary as the user navigates the application. The ViewModels are instantiated as Singleton objects to store the content until the application is closed. The previous configuration will be present when a user return to the page.

11.5.2 Home page



Home

Opc Ua

Connection status: Connected
Security: Sign & Encrypt
Alive: 00:01:57
Connection attemps: 2
Monitored items: 74
Nodes monitored: 2476
Monitor rate[node/s]: 1
Nodes write: 29
Write rate[node/s]: 0
Nodes write errors: 0

Mqtt

Connection status: Connected
Security: TLS v1.2
Alive: 00:24:04
Connection attemps: 1
Subscribed topics: 42
Messages published: 1291
Published rate [msg/s]: 0
Messages received: 38
Receive rate [msg/s]: 0

Debug Opc Ua

NodeId ns=4;i=50
Value True
Value False

Debug Mqtt

Topic UNS/Hvl/WaterRig/LT_01/Operations/ToogleSuppress

Value
```
{
  "DataType": "1",
  "Value": "1"
}
```

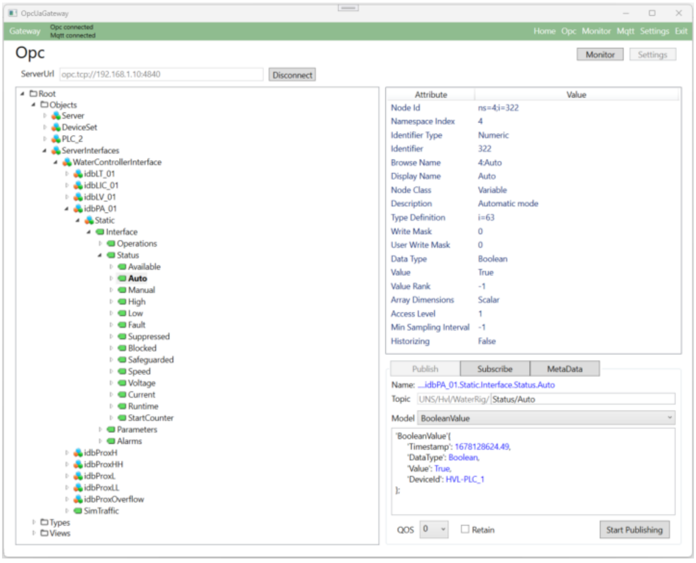
UNS/Hvl/WaterRig/LT_01/Status/Suppressed

Value
```
{
  "Timestamp": "1326615648",
  "DataType": "Boolean",
  "Value": "True",
  "DeviceId": "HVL-PLC_1",
};
```

Figure 130 Gateway application home view

The home page has been designed to give an overview of the connection status and to provide overall statistics of the communication. It also includes a section with a debugger function that can be used for reading or writing to single nodes in the OPC server or to publish or subscribe to topics in the MQTT broker. It has been split in the middle to separate the two communication interfaces.

11.5.3 OPC page



ServerUrl opc.tcp://192.168.1.10:4840

Attribute	Value
Node Id	ns=4;i=322
Namespace Index	4
Identifier Type	Numeric
Identifier	322
Browse Name	4:Auto
Display Name	Auto
Node Class	Variable
Description	Automatic mode
Type Definition	i=63
Write Mask	0
User Write Mask	0
Data Type	Boolean
Value	True
Value Rank	-1
Array Dimensions	Scalar
Access Level	1
Min Sampling Interval	-1
Historizing	False

Name: ....idbPA_01.Static.Interface.Status.Auto

Topic UNS/Hvl/WaterRig/ Status/Auto

Model BooleanValue

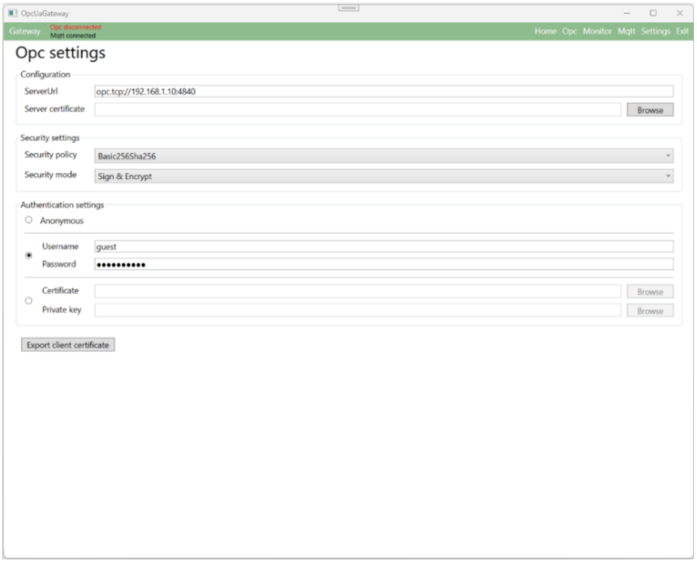
```
'BooleanValue'{
    'Timestamp': 1678128624.49,
    'DataType': Boolean,
    'Value': True,
    'DeviceId': HVL-PLC_1
};
```

QOS 0 ☐ Retain

Figure 131 Gateway application OPC view

The shown page is for connecting and browsing the OPC server address space. Each node expands into a tree view, which you can browse through. Selecting a node will show additional node information and display it in the "attributes list view" on the top right. New mappings are configured online by dragging and dropping the nodes onto the lower right corner of the page, it is then possible to select between publishing or subscribing the node to UNS. Additional metadata is provided depending on the type of model chosen. Available types vary depending on the data type of the node. MQTT settings like the Topic, QoS, and Retain are set before the configuration is implemented. The topic prefix will be fixed to guarantee a relationship to the device on which the OPC server is running on. The MetaData tab has been included for publishing single retain messages consisting of key-value pairs intended to simplify integration for consumers subscribing to data from the device.

On the top right is a button for navigating to the OPC settings tabs for changing the security settings of the OPC session or exporting the client certificate so that it can be imported into the server.



Configuration

ServerUrl opc.tcp://192.168.1.10:4840

Server certificate

Security settings

Security policy Basic256Sha256

Security mode Sign & Encrypt

Authentication settings

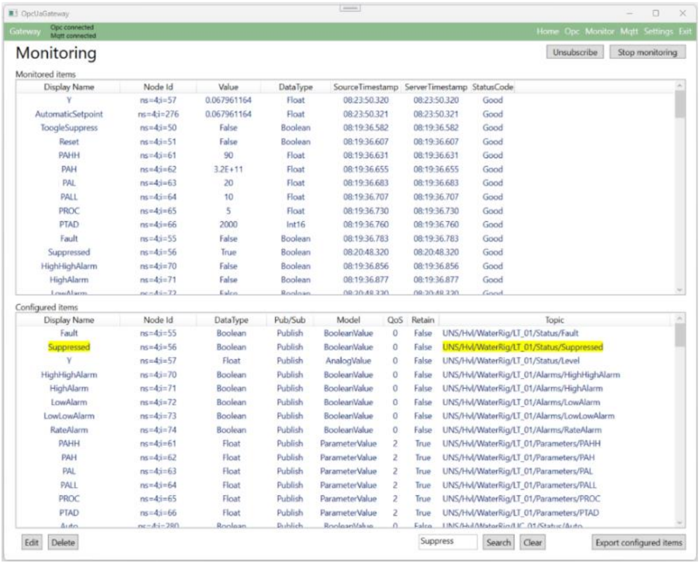
○ Anonymous

● Username guest
Password ••••••••••

○ Certificate
Private key

Figure 132 Gateway application OPC Settings view

11.5.4 Monitor page



Monitoring

Monitored items

Display Name	Node Id	Value	DataType	SourceTimestamp	ServerTimestamp	StatusCode
Y	ns=4;i=57	0.067961164	Float	08:23:50.320	08:23:50.320	Good
AutomaticSetpoint	ns=4;i=276	0.067961164	Float	08:23:50.321	08:23:50.321	Good
ToogleSuppress	ns=4;i=50	False	Boolean	08:19:36.582	08:19:36.582	Good
Reset	ns=4;i=51	False	Boolean	08:19:36.607	08:19:36.607	Good
PAHH	ns=4;i=61	90	Float	08:19:36.631	08:19:36.631	Good
PAH	ns=4;i=62	3.2E+11	Float	08:19:36.655	08:19:36.655	Good
PAL	ns=4;i=63	20	Float	08:19:36.683	08:19:36.683	Good
PALL	ns=4;i=64	10	Float	08:19:36.707	08:19:36.707	Good
PROC	ns=4;i=65	5	Float	08:19:36.730	08:19:36.730	Good
PTAD	ns=4;i=66	2000	Int16	08:19:36.760	08:19:36.760	Good
Fault	ns=4;i=55	False	Boolean	08:19:36.783	08:19:36.783	Good
Suppressed	ns=4;i=56	True	Boolean	08:20:48.320	08:20:48.320	Good
HighHighAlarm	ns=4;i=70	False	Boolean	08:19:36.856	08:19:36.856	Good
HighAlarm	ns=4;i=71	False	Boolean	08:19:36.877	08:19:36.877	Good
LowAlarm	ns=4;i=72	False	Boolean	08:20:48.320	08:20:48.320	Good

Configured items

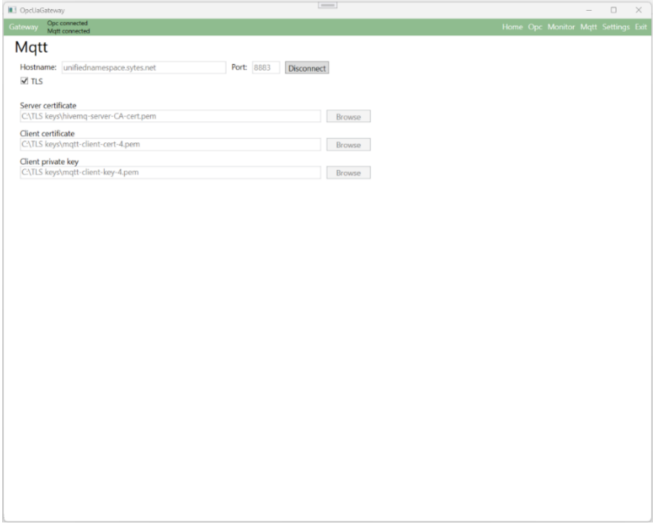
Display Name	Node Id	DataType	Pub/Sub	Model	QoS	Retain	Topic
Fault	ns=4;i=55	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Status/Fault
Suppressed	ns=4;i=56	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Status/Suppressed
Y	ns=4;i=57	Float	Publish	AnalogValue	0	False	UNS/Hvl/WaterRig/LT_01/Status/Level
HighHighAlarm	ns=4;i=70	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Alarms/HighHighAlarm
HighAlarm	ns=4;i=71	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Alarms/HighAlarm
LowAlarm	ns=4;i=72	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Alarms/LowAlarm
LowLowAlarm	ns=4;i=73	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Alarms/LowLowAlarm
RateAlarm	ns=4;i=74	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LT_01/Alarms/RateAlarm
PAHH	ns=4;i=61	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PAHH
PAH	ns=4;i=62	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PAH
PAL	ns=4;i=63	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PAL
PALL	ns=4;i=64	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PALL
PROC	ns=4;i=65	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PROC
PTAD	ns=4;i=66	Float	Publish	ParameterValue	2	True	UNS/Hvl/WaterRig/LT_01/Parameters/PTAD
Auto	ns=4;i=280	Boolean	Publish	BooleanValue	0	False	UNS/Hvl/WaterRig/LIC_01/Status/Auto

Figure 133 Gateway application Monitoring view

The page's top part displays the most recent values of monitored items in the OPC server. The list is implemented with an observable collection to make it dynamic on the UI. The drawback is that it is not possible to search it using hash values. Searching the entire list is required when updating values in the code behind based on recently received messages from the OPC server. It is not a problem as long as the total number of nodes changing every second doesn't exceed approximately 1000 nodes. Monitoring is not essential for mapping between the interfaces and can be switched off if the number increases beyond this limit.

All configured items are listed in the "list view" on the monitoring page. All configured items will be loaded during application startup and added to the list. New configurations made online will also be added. The list can be searched using the toolbar at the bottom of the page. All matches will be highlighted with a yellow background. It is possible to edit or delete existing configurations by selecting them in the list view and using the associated button on the toolbar. A complete list of all configured items can be exported to a .csv file to easily distribute it outside of the application.

11.5.5 MQTT page



Mqtt

Hostname: unifiednamespace.sytes.net Port: 8883

☑ TLS

Server certificate
C:\TLS keys\hivemq-server-CA-cert.pem

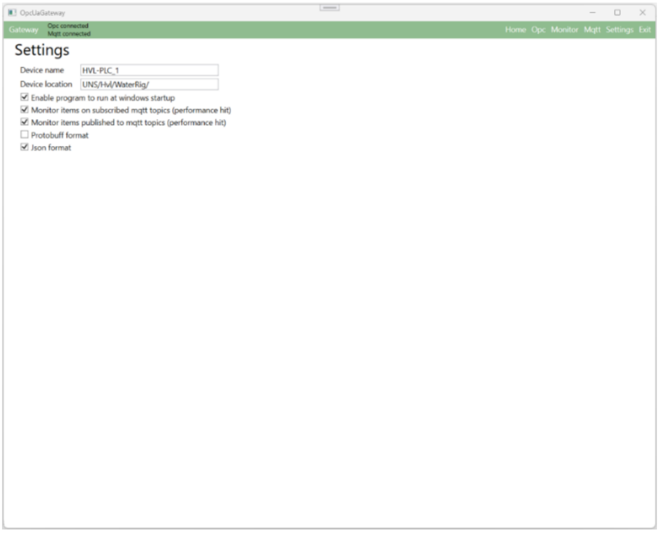
Client certificate
C:\TLS keys\mqtt-client-cert-4.pem

Client private key
C:\TLS keys\mqtt-client-key-4.pem

Figure 134 Gateway application MQTT view

Browsing of TLS certificates used by the client when connecting to the MQTT broker is conducted on the MQTT page. Certificates are verified as valid certificate files before being accepted. The broker address, port number, and the option to connect using TLS are chosen before a connection to the broker is attempted. All settings changed are written to the offline configuration and load when the application restarts.

11.5.6 Settings page



Settings

Device name HVL-PLC_1

Device location UNS/Hvl/WaterRig/

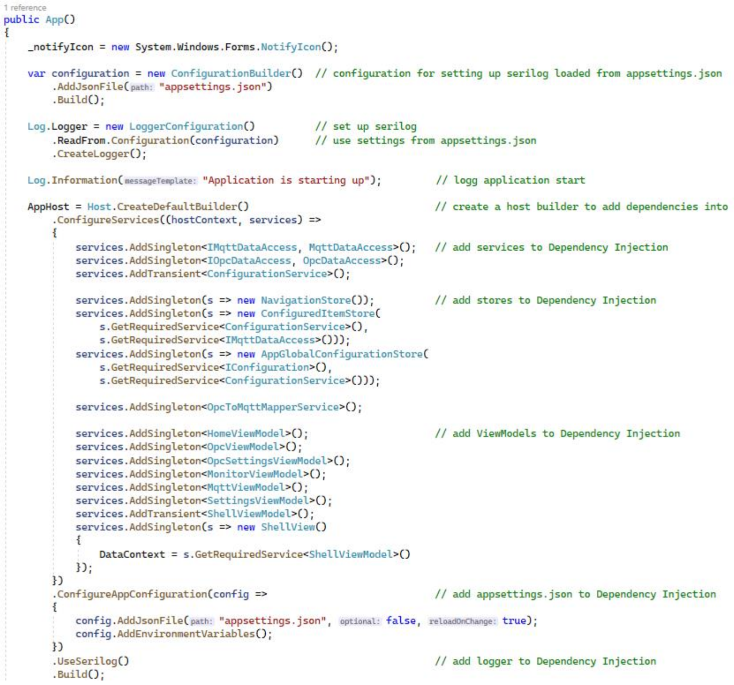
☑ Enable program to run at windows startup
☑ Monitor items on subscribed mqtt topics (performance hit)
☑ Monitor items published to mqtt topics (performance hit)
□ Protobuff format
☑ Json format

Figure 135 Gateway application settings view

Global application settings are changed on this page and will be stored in the offline configuration. The Device name and Device location is dictating the topic prefix and the last will topic. These changes will not alter already configured items. All other settings on this page can be changed while the gateway is in operation and will take effect immediately.

11.6 App.cs

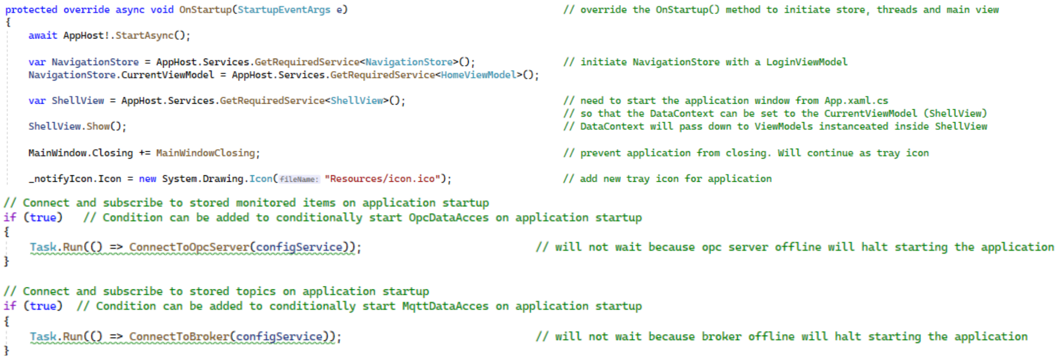
WPF applications are controlled from App.cs, the entry point where classes are instantiated when the app is launched. Here, application services are instantiated before being added to dependency injection. Global application settings like the logger and adding an offline settings file are configured in this class. Objects holding global data that multiple classes need access to are organized into stores. These stores are instantiated as singleton objects (all requesters receive the same object) and added to DI. ViewModels are instantiated as singletons to hold data when the user navigates to a different page. The alternative is to define them as Transients which will provide a fresh ViewModel every time through DI. Instances where it is desired to get clean pages without old settings or search history are examples of when ViewModels should be Transients.



```
1 reference
public App()
{
    _notifyIcon = new System.Windows.Forms.NotifyIcon();

    var configuration = new ConfigurationBuilder()  // configuration for setting up serilog loaded from appsettings.json
        .AddJsonFile(path: "appsettings.json")
        .Build();

    Log.Logger = new LoggerConfiguration()          // set up serilog
        .ReadFrom.Configuration(configuration)      // use settings from appsettings.json
        .CreateLogger();

    Log.Information(messageTemplate: "Application is starting up");         // logg application start

    AppHost = Host.CreateDefaultBuilder()                                   // create a host builder to add dependencies into
        .ConfigureServices((hostContext, services) =>
        {
            services.AddSingleton<IMqttDataAccess, MqttDataAccess>();   // add services to Dependency Injection
            services.AddSingleton<IOpcDataAccess, OpcDataAccess>();
            services.AddTransient<ConfigurationService>();

            services.AddSingleton(s => new NavigationStore());          // add stores to Dependency Injection
            services.AddSingleton(s => new ConfiguredItemStore(
                s.GetRequiredService<ConfigurationService>(),
                s.GetRequiredService<IMqttDataAccess>()));
            services.AddSingleton(s => new AppGlobalConfigurationStore(
                s.GetRequiredService<IConfiguration>(),
                s.GetRequiredService<ConfigurationService>()));

            services.AddSingleton<OpcToMqttMapperService>();

            services.AddSingleton<HomeViewModel>();                     // add ViewModels to Dependency Injection
            services.AddSingleton<OpcViewModel>();
            services.AddSingleton<OpcSettingsViewModel>();
            services.AddSingleton<MonitorViewModel>();
            services.AddSingleton<MqttViewModel>();
            services.AddSingleton<SettingsViewModel>();
            services.AddTransient<ShellViewModel>();
            services.AddSingleton(s => new ShellView()
            {
                DataContext = s.GetRequiredService<ShellViewModel>()
            });
        })
        .ConfigureAppConfiguration(config =>                            // add appsettings.json to Dependency Injection
        {
            config.AddJsonFile(path: "appsettings.json", optional: false, reloadOnChange: true);
            config.AddEnvironmentVariables();
        })
        .UseSerilog()                                                   // add logger to Dependency Injection
        .Build();
```

Figure 136 App.cs including dependency injection

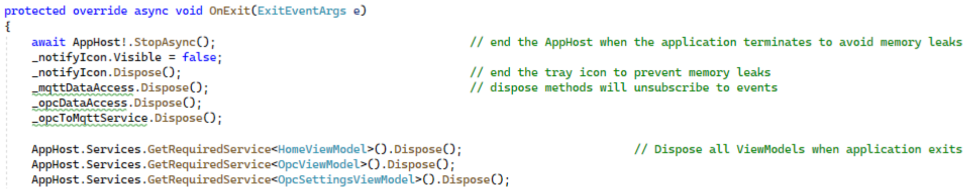
After settings and dependency injection have been configured, the application itself is started. A Shell window with a Home page is displayed to the user and a tray icon is placed on the task bar so that the application can be minimized when the UI is not in use. The DataAccess services are configured to connect at startup and are executed on a different thread. DataAccess is not awaited since it will halt the application startup if the connection to the endpoint is down.



```
protected override async void OnStartup(StartupEventArgs e)                     // override the OnStartup() method to initiate store, threads and main view
{
    await AppHost!.StartAsync();

    var NavigationStore = AppHost.Services.GetRequiredService<NavigationStore>();        // initiate NavigationStore with a LoginViewModel
    NavigationStore.CurrentViewModel = AppHost.Services.GetRequiredService<HomeViewModel>();

    var ShellView = AppHost.Services.GetRequiredService<ShellView>();           // need to start the application window from App.xaml.cs
                                                                                // so that the DataContext can be set to the CurrentViewModel (ShellView)
    ShellView.Show();                                                           // DataContext will pass down to ViewModels instanceated inside ShellView

    MainWindow.Closing += MainWindowClosing;                                    // prevent application from closing. Will continue as tray icon

    _notifyIcon.Icon = new System.Drawing.Icon(fileName: "Resources/icon.ico");  // add new tray icon for application
// Connect and subscribe to stored monitored items on application startup
if (true)   // Condition can be added to conditionally start OpcDataAcces on application startup
{
    Task.Run(() => ConnectToOpcServer(configService));                      // will not wait because opc server offline will halt starting the application
}

// Connect and subscribe to stored topics on application startup
if (true)  // Condition can be added to conditionally start MqttDataAcces on application startup
{
    Task.Run(() => ConnectToBroker(configService));                         // will not wait because broker offline will halt starting the application
}
```

Figure 137 Starting the application and services asynchronously.

App.cs is also the place where the application is terminated. Various services and ViewModels have subscribed to events during the application's lifetime and must be cleaned up to prevent memory leaks. The Dispose method in classes which subscribe to events has been overridden to unsubscribe the event handlers. Then the OnExit method in App.cs is overridden to dispose of the objects.



```
protected override async void OnExit(ExitEventArgs e)
{
    await AppHost!.StopAsync();                             // end the AppHost when the application terminates to avoid memory leaks
    _notifyIcon.Visible = false;
    _notifyIcon.Dispose();                                  // end the tray icon to prevent memory leaks
    _mqttDataAccess.Dispose();                              // dispose methods will unsubscribe to events
    _opcDataAccess.Dispose();
    _opcToMqttService.Dispose();

    AppHost.Services.GetRequiredService<HomeViewModel>().Dispose();                 // Dispose all ViewModels when application exits
    AppHost.Services.GetRequiredService<OpcViewModel>().Dispose();
    AppHost.Services.GetRequiredService<OpcSettingsViewModel>().Dispose();
```

Figure 138 Terminating the application and disposing of services.

11.7 OpcDataAccess class

The class has been implemented without any dependencies and with the interface IOpcDataAccess to achieve loose coupling. It makes the class easily replaceable or reused in other projects. In essence, the class encapsulates OPC UA Services defined by the OPC Foundation into usable methods inside the application. All communication to the OPC server endpoint is performed inside the class and protected from outside interference. All session and subscription variables are inaccessible to outside classes.

The class extensively uses events to communicate back to the caller class, which is unknown until the application is built. Communication between the internal functions of the NetStandard.Opc.Ua library and OpcDataAccess is also dependent on events. Event handlers must be attached to the required events when the class is utilized. The events will relay relevant information to listeners, who will then handle the situation based on what is coded in the event handler.

The garbage collector in C# does not automatically clean up event handlers. The application has been designed and tested for 2000+ monitored items, each attached to a separate event handler. These handlers can remain and take up resources until Windows is restarted, which is unacceptable. It is essential to unregister all the handlers when the class is disposed of. The consequence of not performing event clean-up is that the memory of the host gets exhausted, and it can be problematic to register new handlers when the application has been restarted. Overriding the Dispose method to unregister all events and making sure it is called when the class is disposed of are efficiently solving the issue of dangling events in the operating system.

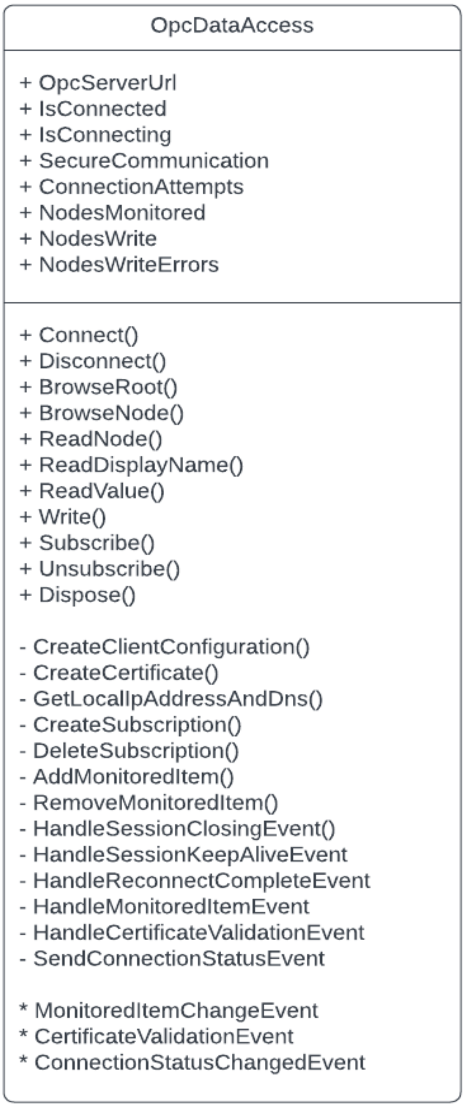
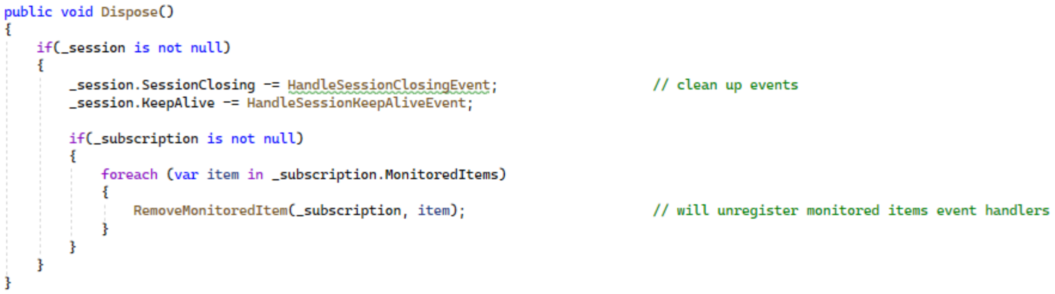


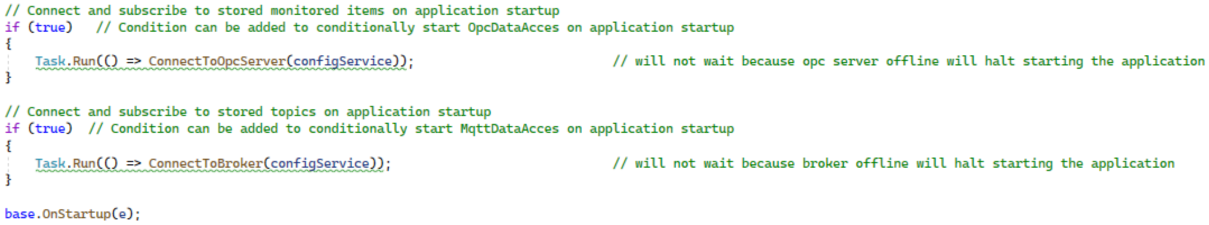
Figure 139 OpcDataAccess class diagram



```
public void Dispose()
{
    if(_session is not null)
    {
        _session.SessionClosing -= HandleSessionClosingEvent;                // clean up events
        _session.KeepAlive -= HandleSessionKeepAliveEvent;

        if(_subscription is not null)
        {
            foreach (var item in _subscription.MonitoredItems)
            {
                RemoveMonitoredItem(_subscription, item);                    // will unregister monitored items event handlers
            }
        }
    }
}
```

Figure 140 Disposing event handlers in OpcDataAccess

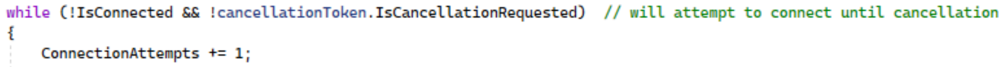
Connecting and maintaining the connection to the OPC server is fundamental to reliable data transfer. It is essential that the gateway has functionality for connection evaluation and can handle a connection that is broken without intervention by the user. The application is configured to connect to the OPC server on startup. An async task will initiate a connection attempt to avoid locking the UI or delaying the application's startup. A spinner on the OPC page will be visible as long as the application is attempting to connect. A new connection attempt is initiated every 10 seconds.



```
// Connect and subscribe to stored monitored items on application startup
if (true)   // Condition can be added to conditionally start OpcDataAcces on application startup
{
    Task.Run(() => ConnectToOpcServer(configService));                         // will not wait because opc server offline will halt starting the application
}

// Connect and subscribe to stored topics on application startup
if (true)  // Condition can be added to conditionally start MqttDataAcces on application startup
{
    Task.Run(() => ConnectToBroker(configService));                             // will not wait because broker offline will halt starting the application
}

base.OnStartup(e);
```

Figure 141 Connecting to OPC and MQTT endpoint at application startup

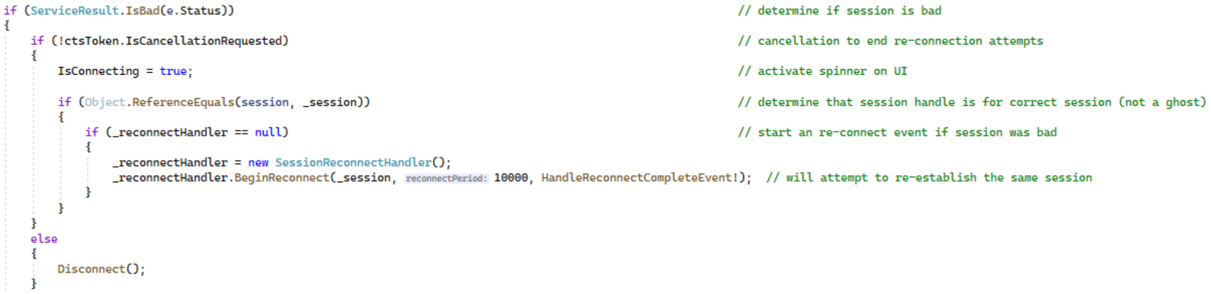
Canceling the connection attempt can be done with a cancellation token which throws an exception inside the connect method. The exception is caught, and no further connection attempts are initiated.



```
while (!IsConnected && !cancellationToken.IsCancellationRequested)  // will attempt to connect until cancellation
{
    ConnectionAttempts += 1;
```

Figure 142 Cancelling current OPC re-connect cycle

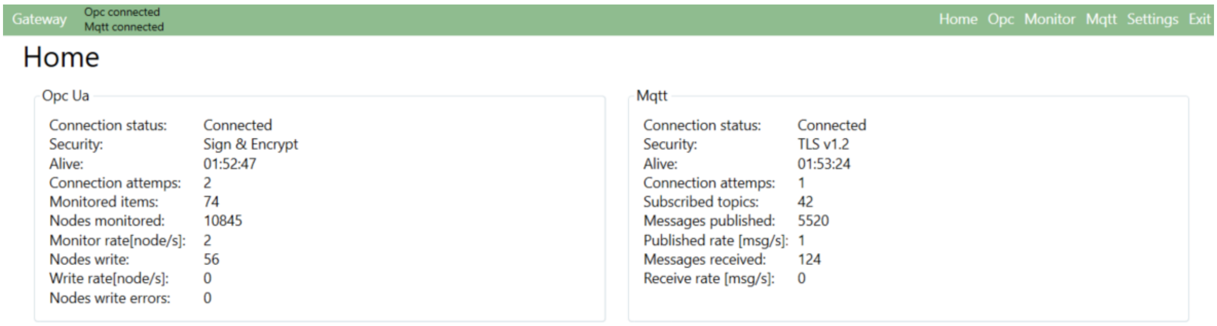
The NetStandard.Opc.Ua library periodically fires an event carrying connection status, which can be evaluated. A reconnect event handler also included in the library is fired if the connection is deemed bad. The handler will continuously attempt to re-establish the previous session if not canceled by the user. A faulty connection is indicated on the UI, and the spinner is turned on as long as the handler attempts to reconnect. The handler must be disposed of when the connection is re-established to avoid additional reconnect requests.



```
if (ServiceResult.IsBad(e.Status))                                                      // determine if session is bad
{
    if (!ctsToken.IsCancellationRequested)                                              // cancellation to end re-connection attempts
    {
        IsConnecting = true;                                                            // activate spinner on UI

        if (Object.ReferenceEquals(session, _session))                                  // determine that session handle is for correct session (not a ghost)
        {
            if (_reconnectHandler == null)                                              // start an re-connect event if session was bad
            {
                _reconnectHandler = new SessionReconnectHandler();
                _reconnectHandler.BeginReconnect(_session, reconnectPeriod: 10000, HandleReconnectCompleteEvent!);  // will attempt to re-establish the same session
            }
        }
    }
    else
    {
        Disconnect();
    }
```

Figure 143 Periodic OPC server connection status evaluation

Logic for keeping track of the connection status and data transfer is included inside the class. The interface defines what statistics are expected to be provided by the DataAccess class. These statistics are read from the client and displayed on the Main page mostly for debugging purposes.



Home

Opc Ua

Connection status:	Connected
Security:	Sign & Encrypt
Alive:	01:52:47
Connection attemps:	2
Monitored items:	74
Nodes monitored:	10845
Monitor rate[node/s]:	2
Nodes write:	56
Write rate[node/s]:	0
Nodes write errors:	0

Mqtt

Connection status:	Connected
Security:	TLS v1.2
Alive:	01:53:24
Connection attemps:	1
Subscribed topics:	42
Messages published:	5520
Published rate [msg/s]:	1
Messages received:	124
Receive rate [msg/s]:	0

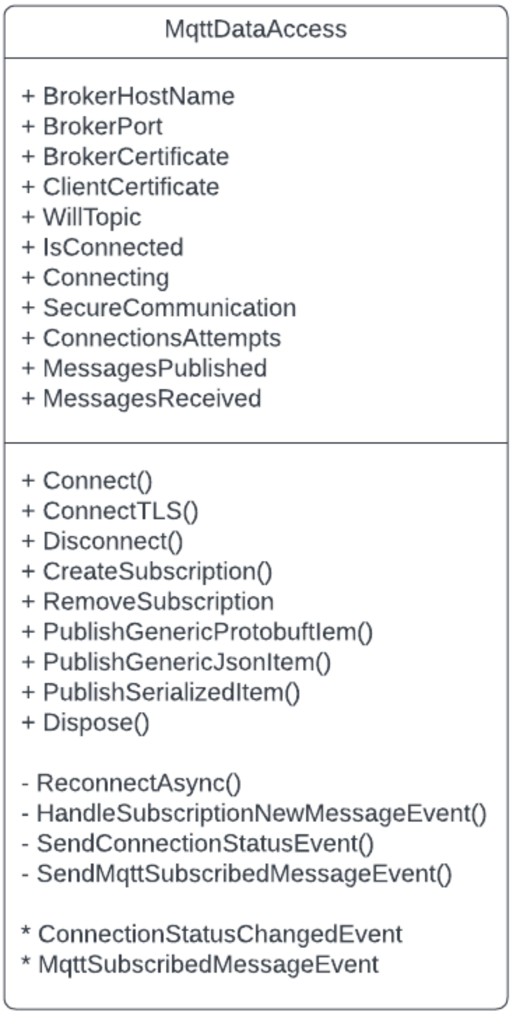
Figure 144 Connection statistics displayed on the home page

11.8 MqttDataAccess class

The MqttDataAccess class is equivalent to OpcDataAccess and contains all the logic for establishing sessions and transmitting publish and subscribe messages on the MQTT interface of the gateway. Statistics are recorded and displayed on the Main page.

The class only has one event for all the topics that will be fired whenever new messages from the broker are received. Another event routes these messages to outside listeners in the application. In addition, the class has an event sent every time the connection status changes. It is used to notify listeners that they must check and update the UI or take action as a result of the connection status being changed.

Almost no restrictions exist on what can be sent in an MQTT message. Any text can be encapsulated. It is up to the receiver to decode it correctly. It can be an advantage that provides flexibility, and a drawback since integration between systems can become more demanding without a locked message format. MqttDataAccess is implementing methods that serialize messages as Json key-value pairs maintaining the flexibility, and other methods which serialized into protobuf encoding based on predefined models. The serialization methods are implemented with generics types to enable any object of type class to be serialized using the same method. The resulting byte array after serialization is handled equally when packed into an MQTT message and published to a topic. A separate class EncodingHelpers is implemented to manage all serialization.



MqttDataAccess

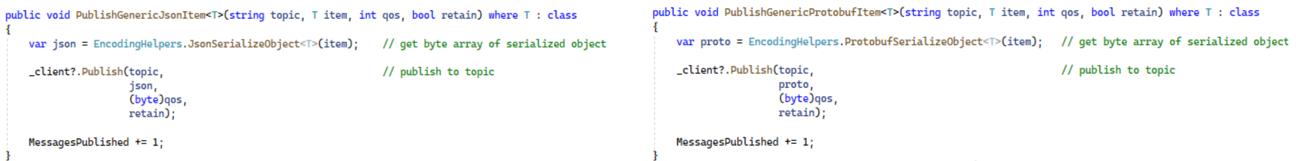
+ BrokerHostName
+ BrokerPort
+ BrokerCertificate
+ ClientCertificate
+ WillTopic
+ IsConnected
+ Connecting
+ SecureCommunication
+ ConnectionsAttempts
+ MessagesPublished
+ MessagesReceived

+ Connect()
+ ConnectTLS()
+ Disconnect()
+ CreateSubscription()
+ RemoveSubscription
+ PublishGenericProtobufItem()
+ PublishGenericJsonItem()
+ PublishSerializedItem()
+ Dispose()

- ReconnectAsync()
- HandleSubscriptionNewMessageEvent()
- SendConnectionStatusEvent()
- SendMqttSubscribedMessageEvent()

* ConnectionStatusChangedEvent
* MqttSubscribedMessageEvent

Figure 146 MqttDataAccess class diagram

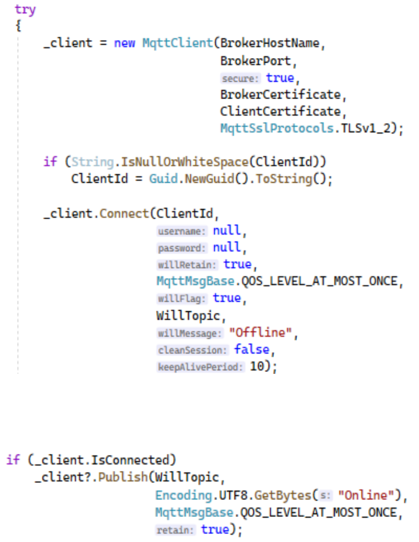


```
public void PublishGenericJsonItem<T>(string topic, T item, int qos, bool retain) where T : class
{
    var json = EncodingHelpers.JsonSerializeObject<T>(item);   // get byte array of serialized object

    _client?.Publish(topic,                                     // publish to topic
                     json,
                     (byte)qos,
                     retain);

    MessagesPublished += 1;
}
```

```
public void PublishGenericProtobufItem<T>(string topic, T item, int qos, bool retain) where T : class
{
    var proto = EncodingHelpers.ProtobufSerializeObject<T>(item);   // get byte array of serialized object

    _client?.Publish(topic,                                          // publish to topic
                     proto,
                     (byte)qos,
                     retain);

    MessagesPublished += 1;
}
```

Figure 145 Methods for encoding and serialize messages into Json or protobuf

Setting the client parameters is important to get the most out of the client and what the MQTT protocol offers. The ClientId and the cleanSession parameters are used to re-establish an old connection. All the session parameters are stored in the broker, and the session is resumed having all configured topics and previous configurations. A will message is communicated to the broker during session establishment. It is used to notify about the device status and will be transmitted to all subscribers if the client goes offline beyond the configured keepAlivePeriod of 10 seconds. The will messages should always be sent as retain messages with a quality of service level of 2 to guarantee that it will be received. The broker will not transmit the last will messages if a client actively disconnects. Therefore, the client must transmit a message to the will topic before disconnection. When the client connects, a message with “Online” status is sent to the will topic to notify subscribers that the device is available.

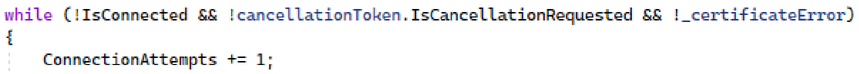


```
try
{
    _client = new MqttClient(BrokerHostName,
                             BrokerPort,
                             secure: true,
                             BrokerCertificate,
                             ClientCertificate,
                             MqttSslProtocols.TLSv1_2);

    if (String.IsNullOrWhiteSpace(ClientId))
        ClientId = Guid.NewGuid().ToString();

    _client.Connect(ClientId,
                    username: null,
                    password: null,
                    willRetain: true,
                    MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE,
                    willFlag: true,
                    WillTopic,
                    willMessage: "Offline",
                    cleanSession: false,
                    keepAlivePeriod: 10);
```

```
if (_client.IsConnected)
    _client?.Publish(WillTopic,
                     Encoding.UTF8.GetBytes(s: "Online"),
                     MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE,
                     retain: true);
```

Figure 147 MQTT client parameters

The same requirements of having a reliable and robust connection also apply to the MQTT client. Continuous connection attempts will be made until the client is connected or canceled by the user. The exception is if there is a TLS certificate error which will not disappear without user intervention. Then the connection attempts will be interrupted, and an error will be visible on the UI. The connection status handler is slightly different than for the OPC class. It will only fire if the event that the connection to the broker has been absent beyond the configured keepAlivePeriod. The event is routed to listeners outside the class that update the UI or perform necessary actions. Internally in MqttDataAccess, the event is used to initiate reconnect attempts which can be aborted at any time using the cancellation token.



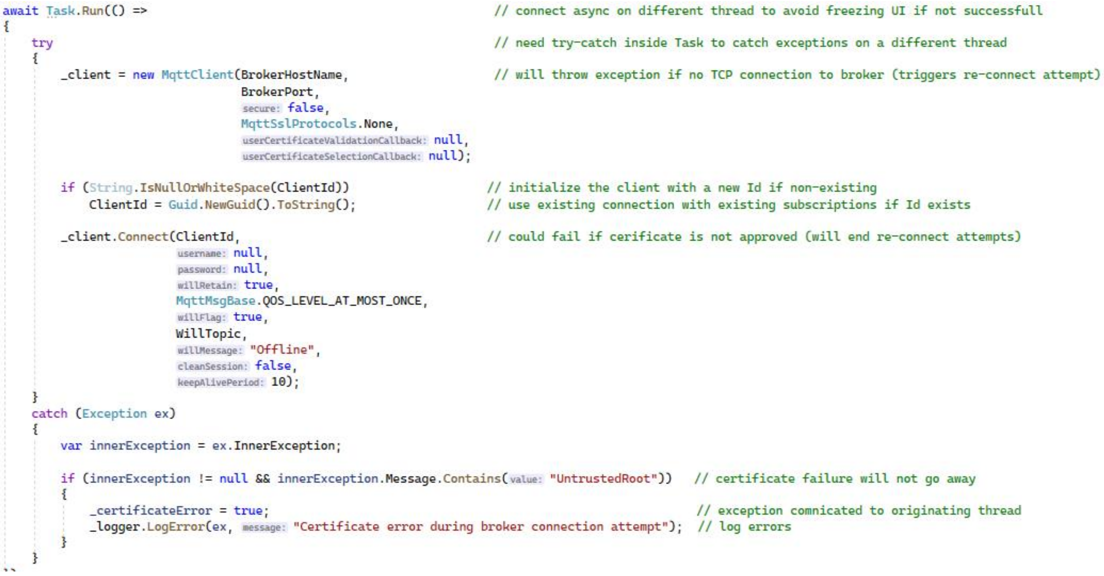
```
while (!IsConnected && !cancellationToken.IsCancellationRequested && !_certificateError)
{
    ConnectionAttempts += 1;
```

Figure 148 Cancelling current MQTT re-connect cycle

11.9 Async wrapping

Connections to remote endpoints can have unpredictable processing times and are often executed asynchronously on separate threads to avoid locking the UI or halting the application. For example, the application processing will freeze until the configured timeout has elapsed if a communication request is sent before a faulty connection is detected by the periodic connection status event handler. Locking the application can be avoided by processing these requests on separate threads. The NetStandard.Opc.Ua has some async methods that can be awaited while a response is pending, but others need to be wrapped as async tasks to ensure the application is never halted.

Any segment of code can be wrapped inside a task for async processing. But an important consideration when tasks are processed on different threads is exceptions thrown inside those tasks which are not caught by the originating thread. Also wrapping these tasks inside try-catch statements and using variables to communicate cross-thread exceptions to enable handling of the exception in the originating thread is essential. Some methods already implement exception handling, but since we have no control or possibility to alter library classes, it is a good practice to wrap all methods on separate threads in try-catch statements.



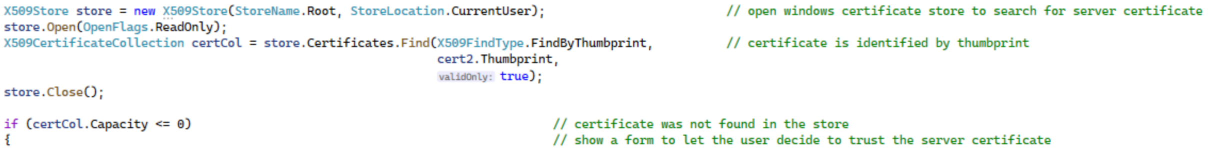
```
await Task.Run(() =>                                                // connect async on different thread to avoid freezing UI if not successfull
{
    try                                                             // need try-catch inside Task to catch exceptions on a different thread
    {
        _client = new MqttClient(BrokerHostName,                    // will throw exception if no TCP connection to broker (triggers re-connect attempt)
                                 BrokerPort,
                                 secure: false,
                                 MqttSslProtocols.None,
                                 userCertificateValidationCallback: null,
                                 userCertificateSelectionCallback: null);

        if (String.IsNullOrWhiteSpace(ClientId))                   // initialize the client with a new Id if non-existing
            ClientId = Guid.NewGuid().ToString();                   // use existing connection with existing subscriptions if Id exists

        _client.Connect(ClientId,                                   // could fail if cerificate is not approved (will end re-connect attempts)
                        username: null,
                        password: null,
                        willRetain: true,
                        MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE,
                        willFlag: true,
                        WillTopic,
                        willMessage: "Offline",
                        cleanSession: false,
                        keepAlivePeriod: 10);
    }
    catch (Exception ex)
    {
        var innerException = ex.InnerException;

        if (innerException != null && innerException.Message.Contains(value: "UntrustedRoot"))   // certificate failure will not go away
        {
            _certificateError = true;                                                           // exception comnicated to originating thread
            _logger.LogError(ex, message: "Certificate error during broker connection attempt");  // log errors
        }
    }
```

Figure 149 Async wrapping of methods into Tasks.

11.10 Cryptography

The UNS architecture is configured only to allow encrypted connections between clients and the MQTT broker cluster. The only OPC server interface at the local network at Hvl is configured with Basic256Sha256 using Sign&Encrypt. Both connections require X509 certificates to authenticate and encrypt messages. It is Therefore required that the application can import, store, and verify X509 certificates.

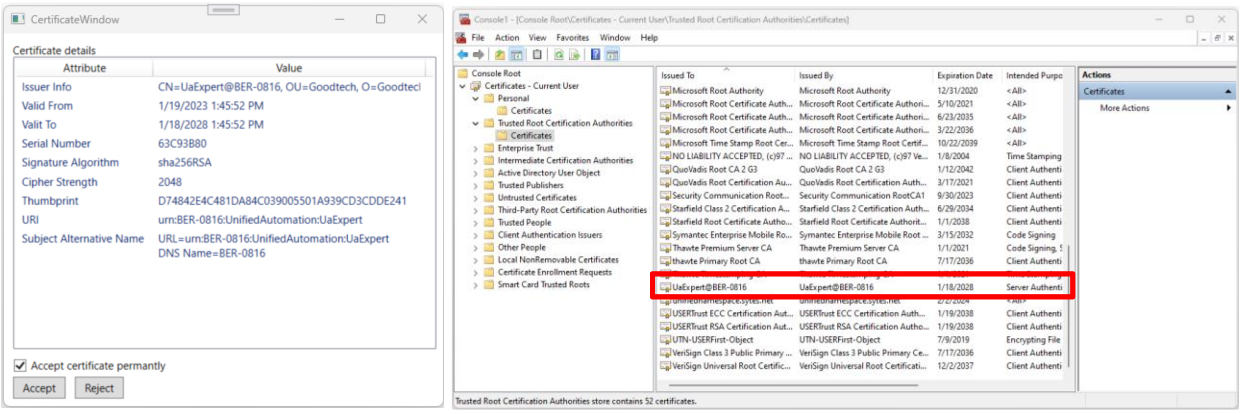
The clients need to know about the server's certificate, which is loaded into the client during initialization and used when a session is established. In addition, certain installations of Windows can refuse the connection if the certificate does not have a trusted root. The same warning appears on HTTPS pages if the certificate has expired or lacks a signature from a trusted CA. To prevent windows from closing the connections, the gateway application will prompt the user to add the server certificate to windows certificate store if it does not already exist there. The result is the same as if the user had found the certificate in windows explorer, opened it, and selected to install the certificate.



```
X509Store store = new X509Store(StoreName.Root, StoreLocation.CurrentUser);            // open windows certificate store to search for server certificate
store.Open(OpenFlags.ReadOnly);
X509CertificateCollection certCol = store.Certificates.Find(X509FindType.FindByThumbprint,          // certificate is identified by thumbprint
                                                           cert2.Thumbprint,
                                                           validOnly: true);

store.Close();

if (certCol.Capacity <= 0)                                                    // certificate was not found in the store
{                                                                             // show a form to let the user decide to trust the server certificate
```

Figure 150 Locating OPC server certificate in windows certificate store

A new certificate will trigger a window where details are shown to the user, who then has the choice to approve or reject the certificate. The certificate is stored permanently in the Windows Store if the user selects "Accept certificate permanently." If the certificate has been added to Windows Store, it has been trusted, and future connections will be considered authenticated.



Attribute	Value
Issuer Info	CN=UaExpert@BER-0816, OU=Goodtech, O=Goodtecl
Valid From	1/19/2023 1:45:52 PM
Valit To	1/18/2028 1:45:52 PM
Serial Number	63C93B80
Signature Algorithm	sha256RSA
Cipher Strength	2048
Thumbprint	D74842E4C481DA84C039005501A939CD3CDDE241
URI	urn:BER-0816:UnifiedAutomation:UaExpert
Subject Alternative Name	URL=urn:BER-0816:UnifiedAutomation:UaExpert DNS Name=BER-0816

☑ Accept certificate permantly

Accept
Reject
Console1 - [Console Root\Certificates - Current User\Trusted Root Certification Authorities\Certificates]
File
Action
View
Favorites
Window
Help
Console Root
Certificates - Current User
Personal
Certificates
Trusted Root Certification Authorities
Certificates
Enterprise Trust
Intermediate Certification Authorities
Active Directory User Object
Trusted Publishers
Untrusted Certificates
Third-Party Root Certification Authorities
Trusted People
Client Authentication Issuers
Other People
Local NonRemovable Certificates
Certificate Enrollment Requests
Smart Card Trusted Roots
Actions
Certificates
More Actions

Issued To	Issued By	Expiration Date	Intended Purpo
Microsoft Root Authority	Microsoft Root Authority	12/31/2020	<All>
Microsoft Root Certificate Auth...	Microsoft Root Certificate Authori...	5/10/2021	<All>
Microsoft Root Certificate Auth...	Microsoft Root Certificate Authori...	6/23/2035	<All>
Microsoft Root Certificate Auth...	Microsoft Root Certificate Authori...	3/22/2036	<All>
Microsoft Time Stamp Root Cer...	Microsoft Time Stamp Root Certif...	10/22/2039	<All>
NO LIABILITY ACCEPTED, (c)97 ...	NO LIABILITY ACCEPTED, (c)97 Ve...	1/8/2004	Time Stamping
QuoVadis Root CA 2 G3	QuoVadis Root CA 2 G3	1/12/2042	Client Authenti
QuoVadis Root Certification Au...	QuoVadis Root Certification Auth...	3/17/2021	Client Authenti
Security Communication Root...	Security Communication RootCA1	9/30/2023	Client Authenti
Starfield Class 2 Certification A...	Starfield Class 2 Certification Auth...	6/29/2034	Client Authenti
Starfield Root Certificate Autho...	Starfield Root Certificate Authorit...	1/1/2038	Client Authenti
Symantec Enterprise Mobile Ro...	Symantec Enterprise Mobile Root ...	3/15/2032	Code Signing
Thawte Premium Server CA	Thawte Premium Server CA	1/1/2021	Code Signing, S
thawte Primary Root CA	thawte Primary Root CA	7/17/2036	Client Authenti
Thawte Timestamping CA	Thawte Timestamping CA	1/1/2021	Time Stamping
UaExpert@BER-0816	UaExpert@BER-0816	1/18/2028	Server Authenti
unifiednamespace.sytes.net	unifiednamespace.sytes.net	2/2/2024	<All>
USERTrust ECC Certification Aut...	USERTrust ECC Certification Auth...	1/19/2038	Client Authenti
USERTrust RSA Certification Aut...	USERTrust RSA Certification Autho...	1/19/2038	Client Authenti
UTN-USERFirst-Object	UTN-USERFirst-Object	7/9/2019	Encrypting File
VeriSign Class 3 Public Primary ...	VeriSign Class 3 Public Primary Ce...	7/17/2036	Client Authenti
VeriSign Universal Root Certific...	VeriSign Universal Root Certificati...	12/2/2037	Client Authenti

Trusted Root Certification Authorities store contains 52 certificates.

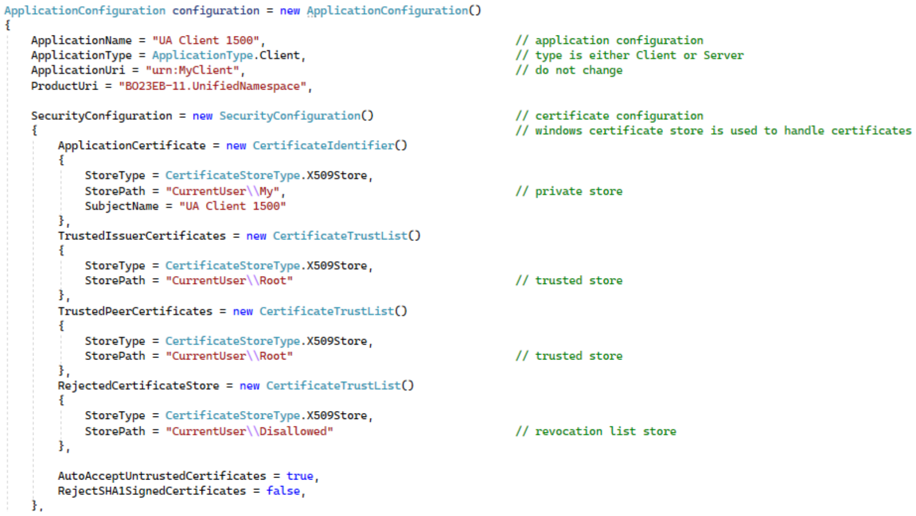
Figure 151 Accepting and adding server certificate into windows certificate store (right side)

Securing the traffic on the HVL LAN will be independent of the broker connection. Separate certificates are used for the different connections to further increase security by adding defense in depth. The two clients are based on two different libraries providing separate levels of security. M2MqttDotNetCore library is lightweight, providing only core services to handle the broker connection, publish, and subscribe requests. There is the possibility to use X509 certificates for TLS connections, but all logic must be programmed outside of the library. NetStandard.OPC.UA library is a comprehensive library with many service functions, including handling of certificates and interaction with windows certificate store. Consequently, the two clients handle certificates differently.

11.10.1 Certificate handling in OpcDataAccess

The NetStandard.OPC.UA library depends on Windows certificate store to handle all certificates. The different sub-stores are referenced by the OPC client during instantiation. This way of handling certificates provides a great advantage since the exposure of certificates by untested third-party code is avoided, and security is based on Microsoft's well-proven code. Three stores are used to validate certificates during runtime.

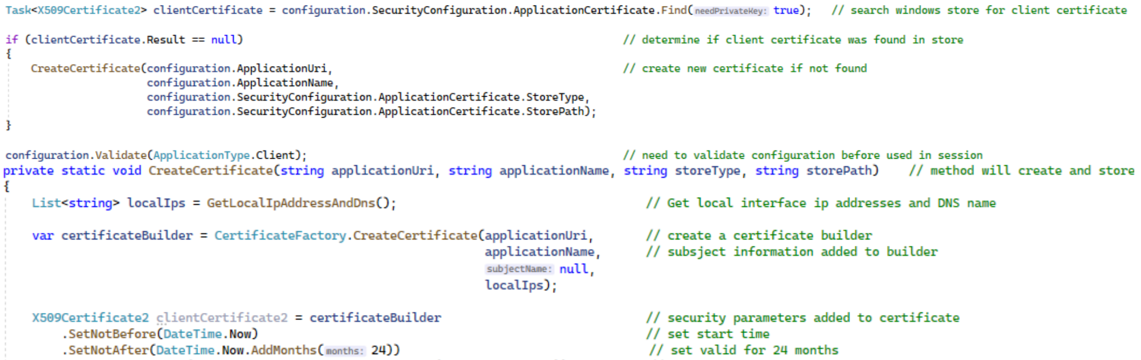
- Personal stores are used to store the client certificate. A new client certificate will be created and put into this store if the client has no certificate on instantiation.
- Trusted root certification authorities are used to store the certificates of trusted devices like the server certificate.
- Untrusted Certificates maintains a list of certificates for devices not trusted. A device certificate can end up in this store if the authentication process fails during session establishment. If so, it then has to be removed manually.



```
ApplicationConfiguration configuration = new ApplicationConfiguration()
{
    ApplicationName = "UA Client 1500",                                      // application configuration
    ApplicationType = ApplicationType.Client,                                // type is either Client or Server
    ApplicationUri = "urn:MyClient",                                         // do not change
    ProductUri = "BO23EB-11.UnifiedNamespace",

    SecurityConfiguration = new SecurityConfiguration()                      // certificate configuration
    {                                                                        // windows certificate store is used to handle certificates
        ApplicationCertificate = new CertificateIdentifier()
        {
            StoreType = CertificateStoreType.X509Store,
            StorePath = "CurrentUser\\My",                                   // private store
            SubjectName = "UA Client 1500"
        },
        TrustedIssuerCertificates = new CertificateTrustList()
        {
            StoreType = CertificateStoreType.X509Store,
            StorePath = "CurrentUser\\Root"                                  // trusted store
        },
        TrustedPeerCertificates = new CertificateTrustList()
        {
            StoreType = CertificateStoreType.X509Store,
            StorePath = "CurrentUser\\Root"                                  // trusted store
        },
        RejectedCertificateStore = new CertificateTrustList()
        {
            StoreType = CertificateStoreType.X509Store,
            StorePath = "CurrentUser\\Disallowed"                            // revocation list store
        },

        AutoAcceptUntrustedCertificates = true,
        RejectSHA1SignedCertificates = false,
    },
```

Figure 152 Certificate stores used in the gateway Opc client

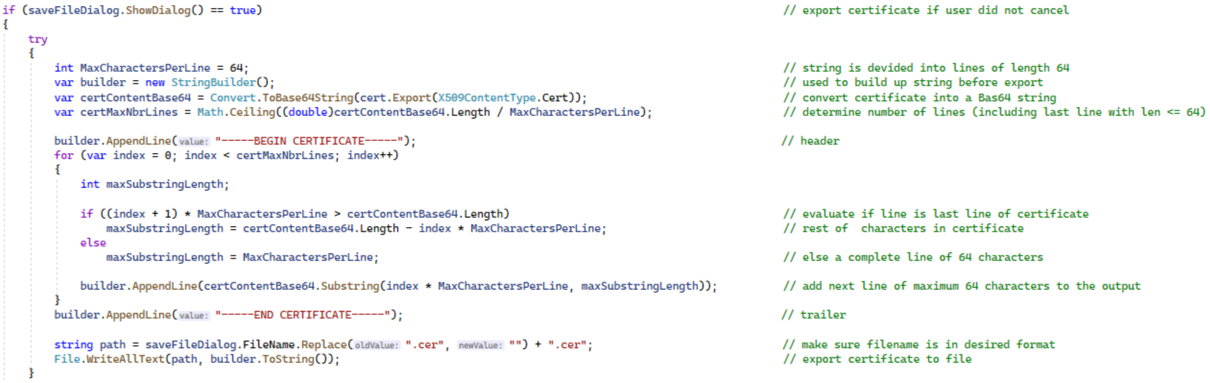
Personal directory inside windows certificate store is checked for a certificate with the subject name "UA Client 1500" as part of the client configuration and will be imported to the client if it exists. If it does not exist, a new certificate will be created by reading the IP addresses and DNS names of the host which the client is running on. These subject names will be used together with the security parameter to generate a new certificate stored in the windows store. The certificate must be exported from the client and imported into the OPC UA server in the TIA portal as a trusted device certificate before communication will be accepted by the server.



```
Task<X509Certificate2> clientCertificate = configuration.SecurityConfiguration.ApplicationCertificate.Find(needPrivateKey: true);   // search windows store for client certificate

if (clientCertificate.Result == null)                                                  // determine if client certificate was found in store
{
    CreateCertificate(configuration.ApplicationUri,                                    // create new certificate if not found
                      configuration.ApplicationName,
                      configuration.SecurityConfiguration.ApplicationCertificate.StoreType,
                      configuration.SecurityConfiguration.ApplicationCertificate.StorePath);
}

configuration.Validate(ApplicationType.Client);                                        // need to validate configuration before used in session
private static void CreateCertificate(string applicationUri, string applicationName, string storeType, string storePath)    // method will create and store
{
    List<string> localIps = GetLocalIpAddressAndDns();                                   // Get local interface ip addresses and DNS name

    var certificateBuilder = CertificateFactory.CreateCertificate(applicationUri,        // create a certificate builder
                                                                  applicationName,       // subsject information added to builder
                                                                  subjectName: null,
                                                                  localIps);

    X509Certificate2 clientCertificate2 = certificateBuilder                             // security parameters added to certificate
        .SetNotBefore(DateTime.Now)                                                      // set start time
        .SetNotAfter(DateTime.Now.AddMonths(months: 24))                                 // set valid for 24 months
```

Figure 153 Creating a new certificate for the Gateway Opc client

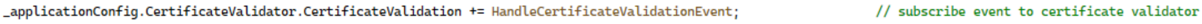
Exporting the client certificate can be done on the OpcSettings page by pressing Export client certificate Then the certificate will be retrieved from the windows store and written to a .cer file which can be imported directly into the TIA portal. The result is the same as going into mmc.exe and manually exporting the certificate in windows.



```
if (saveFileDialog.ShowDialog() == true)                                                         // export certificate if user did not cancel
{
    try
    {
        int MaxCharactersPerLine = 64;                                                           // string is devided into lines of length 64
        var builder = new StringBuilder();                                                       // used to build up string before export
        var certContentBase64 = Convert.ToBase64String(cert.Export(X509ContentType.Cert));      // convert certificate into a Bas64 string
        var certMaxNbrLines = Math.Ceiling((double)certContentBase64.Length / MaxCharactersPerLine); // determine number of lines (including last line with len <= 64)

        builder.AppendLine(value: "-----BEGIN CERTIFICATE-----");                                 // header
        for (var index = 0; index < certMaxNbrLines; index++)
        {
            int maxSubstringLength;

            if ((index + 1) * MaxCharactersPerLine > certContentBase64.Length)                   // evaluate if line is last line of certificate
                maxSubstringLength = certContentBase64.Length - index * MaxCharactersPerLine;    // rest of  characters in certificate
            else
                maxSubstringLength = MaxCharactersPerLine;                                       // else a complete line of 64 characters

            builder.AppendLine(certContentBase64.Substring(index * MaxCharactersPerLine, maxSubstringLength)); // add next line of maximum 64 characters to the output
        }
        builder.AppendLine(value: "-----END CERTIFICATE-----");                                   // trailer

        string path = saveFileDialog.FileName.Replace(oldValue: ".cer", newValue: "") + ".cer";   // make sure filename is in desired format
        File.WriteAllText(path, builder.ToString());                                             // export certificate to file
    }
```

Figure 154 Exporting Gateway OPC client certificate

The server certificate can either be imported on the OpcSettings page or received during session establishment and accepted, thereby adding it to the windows store for future use. A handler method needs to be attached to a CertificateValidation event before the sessions are created. It will fire in the event of an unknown server certificate and prompt the user. One must be aware that if the certificate is not trusted, it will end up in the untrusted store and must be manually deleted from there before a new prompt is given to the user.

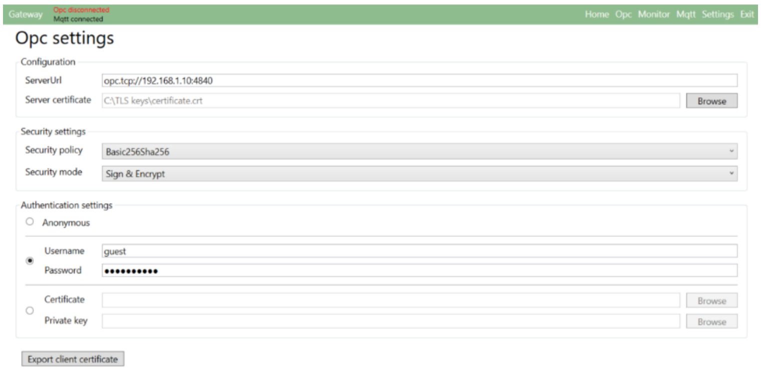


```
_applicationConfig.CertificateValidator.CertificateValidation += HandleCertificateValidationEvent;          // subscribe event to certificate validator
```

Figure 155 OPC client certificate handler used to fetch server certificate

The OpcDataAccess class is intended to be a generic library class and should not include hardcoded handling of the certificate event. The event is therefore routed to listeners outside of OpcDataAccess class.

Mandatory OPC security settings are set through the OpcSettings page and stored in appsettings.json to maintain the configurations when the application restarts. When a new session is established, these settings are passed to the connect method.



Opc settings

Configuration

ServerUrl: opc.tcp://192.168.1.10:4840

Server certificate: C:\TLS keys\certificate.crt [Browse]

Security settings

Security policy: Basic256Sha256

Security mode: Sign & Encrypt

Authentication settings

○ Anonymous

● Username: guest
Password: ••••••••••

○ Certificate: [Browse]
Private key: [Browse]

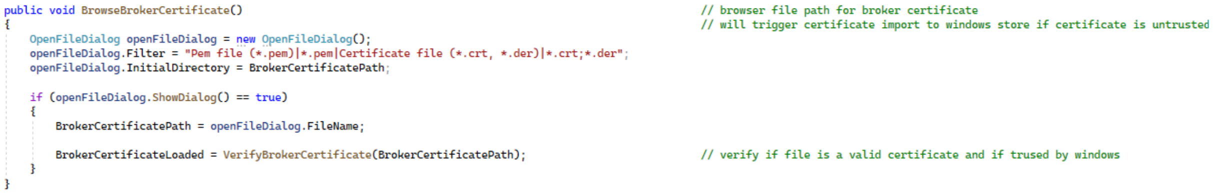
[Export client certificate]

Figure 156 OPC settings view for configuring Opc security

The Siemens PLC is not implementing authentication by certificates. The option to browse these certificates has been templated on the UI, but the logic for passing them to the connect method does not exist since it requires an OPC server endpoint to test against.

11.10.2 Certificates handling in MqttDataAccess

All handling of certificates is done outside of MqttDataAccess class. Certificates are browsed and validated to ensure they are in a legal file format before being stored in local variables inside the client class. The class CertificateHelpers has been implemented to load the certificates and validate different file formats. These are then passed to the connect method if TLS is selected for the MQTT client.

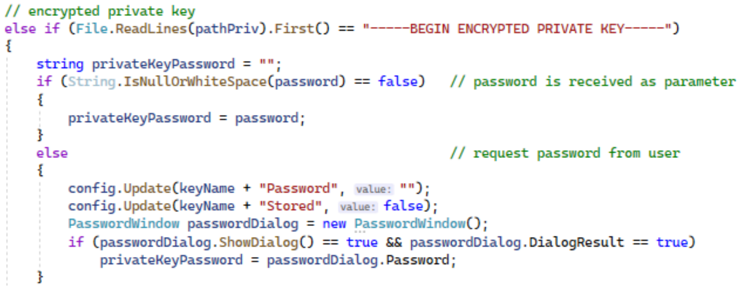


```
public void BrowseBrokerCertificate()                                                // browser file path for broker certificate
{                                                                                    // will trigger certificate import to windows store if certificate is untrusted
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.Filter = "Pem file (*.pem)|*.pem|Certificate file (*.crt, *.der)|*.crt;*.der";
    openFileDialog.InitialDirectory = BrokerCertificatePath;

    if (openFileDialog.ShowDialog() == true)
    {
        BrokerCertificatePath = openFileDialog.FileName;

        BrokerCertificateLoaded = VerifyBrokerCertificate(BrokerCertificatePath);    // verify if file is a valid certificate and if trused by windows
    }
}
```

Figure 157 Browsing for MQTT certificates

The broker certificate is verified to ensure that it is trusted by windows. The same prompt as OpcDataAccess class with the option of adding it to windows store will be presented to the user if an unknown certificate is loaded.

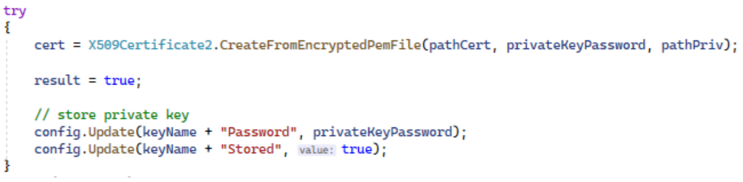
Client certificates are checked for a private key or combined with one to create a file format compatible with the method of connection. Client certificate chains are not verified since it is the server that needs to trust these. Private keys and combined certificates can be password protected. The CertificateHelper methods will prompt the user for a password if required and store it in appsettings.json to enable automatic connection during application startup.



```
// encrypted private key
else if (File.ReadLines(pathPriv).First() == "-----BEGIN ENCRYPTED PRIVATE KEY-----")
{
    string privateKeyPassword = "";
    if (String.IsNullOrWhiteSpace(password) == false)   // password is received as parameter
    {
        privateKeyPassword = password;
    }
    else                                                // request password from user
    {
        config.Update(keyName + "Password", value: "");
        config.Update(keyName + "Stored", value: false);
        PasswordWindow passwordDialog = new PasswordWindow();
        if (passwordDialog.ShowDialog() == true && passwordDialog.DialogResult == true)
            privateKeyPassword = passwordDialog.Password;
    }
```

Figure 158 Verifying MQTT client certificates with passwords from storage or user input

Storing the private key passwords in app settings compromises security because it becomes visible to anyone with access to the application directory. This decision has been made on the assumption that the host where the gateway is running will not be accessible to people who should not have access to the private key password. When the program is distributed, stored passwords will be specific to the certificate being imported and only within reach of those with local access to the host, which should be protected by windows login.



```
try
{
    cert = X509Certificate2.CreateFromEncryptedPemFile(pathCert, privateKeyPassword, pathPriv);

    result = true;

    // store private key
    config.Update(keyName + "Password", privateKeyPassword);
    config.Update(keyName + "Stored", value: true);
}
```

Figure 159 Storing MQTT client certificate into application configuration

11.11 OpcToMqttMapperService

The mapper service is not implemented as a library function. It requires dependencies to the DataAccess classes and global application storage to route messages between the two interfaces.

The purpose of the class is to subscribe to monitored items and subscribed topics events and do a filtering of what messages to pass between the interfaces according to a list of configured items. Additional metadata is added to the messages according to the stored configuration.

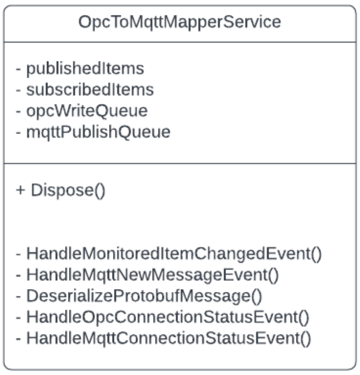
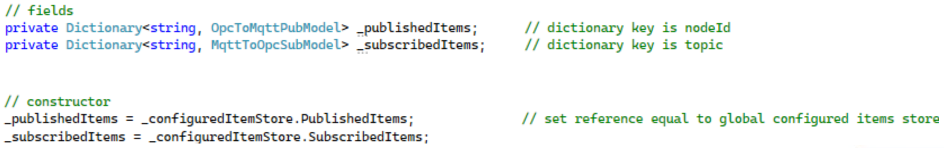

OpcToMqttMapperService
- publishedItems
- subscribedItems
- opcWriteQueue
- mqttPublishQueue
+ Dispose()
- HandleMonitoredItemChangedEvent()
- HandleMqttNewMessageEvent()
- DeserializeProtobufMessage()
- HandleOpcConnectionStatusEvent()
- HandleMqttConnectionStatusEvent()

Figure 160 OpcToMqtt mapper class diagram

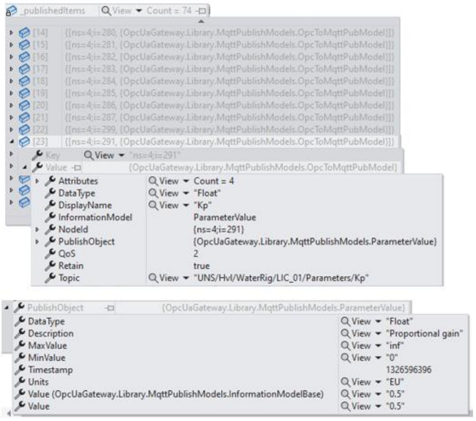
The list of configured items needs to be searched every time a new message or monitored event is fired to determine if it should be mapped to the other interface. The searching through the configured items needs to be fast to guarantee a high throughput. A dictionary object is therefore used to store the runtime configuration. The key is the same as the topic for subscribed items and the node id for monitored items to avoid extra conversions.



```
// fields
private Dictionary<string, OpcToMqttPubModel> _publishedItems;      // dictionary key is nodeId
private Dictionary<string, MqttToOpcSubModel> _subscribedItems;     // dictionary key is topic


// constructor
_publishedItems = _configuredItemStore.PublishedItems;             // set reference equal to global configured items store
_subscribedItems = _configuredItemStore.SubscribedItems;
```

Figure 161 Dictionaries with hash keys used for efficient lookups.

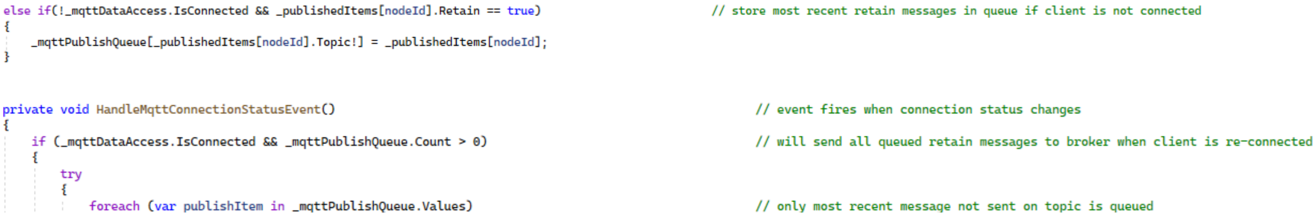
The reference to the configured object will be determined with O(1) lookups. The configured object class is designed to hold all the additional information except for what is provided in the event required to publish a new message on one of the interfaces. Organizing data in one object enables all metadata to be obtained using O(1) operations. The amount of metadata is therefore not the determining factor that will limit the throughput. Updating the metadata is also done once and then published every time with the object.



```
_publishedItems    View ▾ Count = 74
  [14]   {[ns=4;i=280, {OpcUaGateway.Library.MqttPublishModels.OpcToMqttPubModel}]}
  [15]   {[ns=4;i=281, {OpcUaGateway.Library.MqttPublishModels.OpcToMqttPubModel}]}
  [16]   {[ns=4;i=282, {OpcUaGateway.Library.MqttPublishModels.OpcToMqttPubModel}]}
  [17]   {[ns=4;i=283, {OpcUaGateway.Library.MqttPublishModels.OpcToMqttPubModel}]}
  [18]   {[ns=4;i=284, {OpcUaGateway.Library.MqttPublishModels.OpcToMqttPubModel}]}
  [19]   {[ns=4;i=285, {OpcUaGateway.Library.MqttPublishModels.OpcToMqttPubModel}]}
  [20]   {[ns=4;i=286, {OpcUaGateway.Library.MqttPublishModels.OpcToMqttPubModel}]}
  [21]   {[ns=4;i=287, {OpcUaGateway.Library.MqttPublishModels.OpcToMqttPubModel}]}
  [22]   {[ns=4;i=299, {OpcUaGateway.Library.MqttPublishModels.OpcToMqttPubModel}]}
  [23]   {[ns=4;i=291, {OpcUaGateway.Library.MqttPublishModels.OpcToMqttPubModel}]}
    Key      View ▾ "ns=4;i=291"
    Value                {OpcUaGateway.Library.MqttPublishModels.OpcToMqttPubModel}
      Attributes         View ▾ Count = 4
      DataType           View ▾ "Float"
      DisplayName        View ▾ "Kp"
      InformationModel          ParameterValue
      NodeId                    {ns=4;i=291}
      PublishObject             {OpcUaGateway.Library.MqttPublishModels.ParameterValue}
      QoS                       2
      Retain                    true
      Topic              View ▾ "UNS/Hvl/WaterRig/LIC_01/Parameters/Kp"

PublishObject              {OpcUaGateway.Library.MqttPublishModels.ParameterValue}
  DataType                                                         View ▾ "Float"
  Description                                                      View ▾ "Proportional gain"
  MaxValue                                                         View ▾ "inf"
  MinValue                                                         View ▾ "0"
  Timestamp                                                               1326596396
  Units                                                            View ▾ "EU"
  Value (OpcUaGateway.Library.MqttPublishModels.InformationModelBase)  View ▾ "0.5"
  Value                                                            View ▾ "0.5"
```

Figure 162 meta data stored on the objects.

The gateway has two interfaces that can be connected independently of each other. Messages defined as retain by the MQTT protocol will be seen as delivered as long as confirmation is received from the MQTT client and OPC monitored items have no delivery guarantee. Internal logic to set up a queue for messages when one of the interfaces is offline has therefore been implemented. The client connection status is evaluated, and the retain messages are queued if one of the clients are not connected.



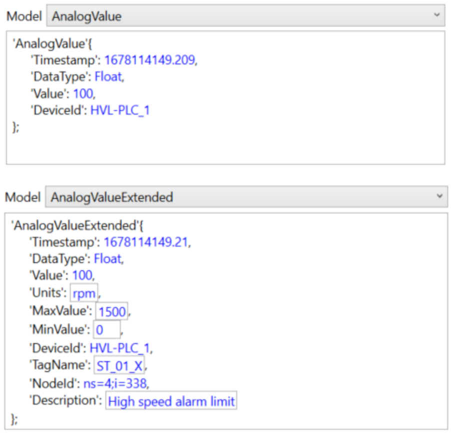
```
else if(!_mqttDataAccess.IsConnected && _publishedItems[nodeId].Retain == true)          // store most recent retain messages in queue if client is not connected
{
    _mqttPublishQueue[_publishedItems[nodeId].Topic!] = _publishedItems[nodeId];
}


private void HandleMqttConnectionStatusEvent()                                        // event fires when connection status changes
{
    if (_mqttDataAccess.IsConnected && _mqttPublishQueue.Count > 0)                   // will send all queued retain messages to broker when client is re-connected
    {
        try
        {
            foreach (var publishItem in _mqttPublishQueue.Values)                     // only most recent message not sent on topic is queued
```

Figure 163 Temporary storing MQTT messages while endpoint connection is offline.

11.12 Information modeling

The MQTT standard does not enforce restrictions on the information contained inside messages which can be a drawback since no contract between sender and receiver guarantees recognizable data structures or compatible data types. Information has been modeled as objects with predefined attributes and data types to overcome this shortcoming. These models can then be serialized as Json or Protobuf encoded messages. Protobuf has the advantage of putting strict enforcement on order and the type of attributes, making it possible to distribute the model blueprint to the receiver beforehand. Then only values can be transmitted with a fixed length int identifier for each attribute. A string identifier requires 8 bits per char in comparison. Protobuf also binary encodes the values since the blueprint dictates the data type. Alternatively, Json key-value pairs can be used to provide the flexibility of not enforcing the data structure by transmitting all messages as strings but at an increased cost of bandwidth and predictability.

One of the application's predefined models must be used when new mappings between the OPC interface and the MQTT broker are to be configured. There are models made with few attributes and minimal metadata well suited for data that is updated and transmitted frequently. And others have extended metadata to provide increased context and are well suited for data such as parameters which are only updated and sent when the user manually makes changes. Using the models guarantees that the values are in the expected and compatible format. The fields the user is allowed to fill in are mostly for metadata having the same datatype as the value attribute or strings, which only adds to the context.

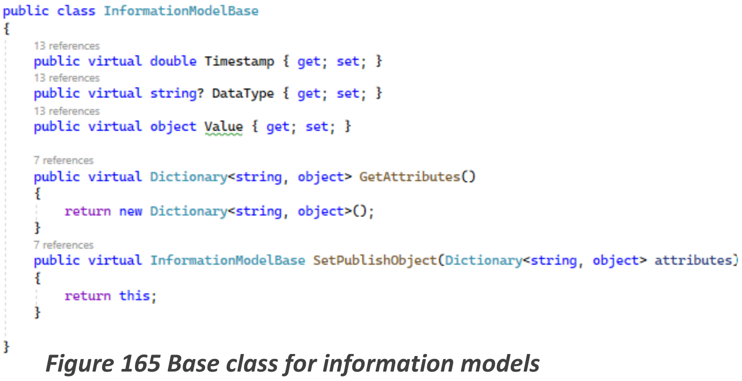


```
'AnalogValue'{
    'Timestamp': 1678114149.209,
    'DataType': Float,
    'Value': 100,
    'DeviceId': HVL-PLC_1
};
```

```
'AnalogValueExtended'{
    'Timestamp': 1678114149.21,
    'DataType': Float,
    'Value': 100,
    'Units': rpm,
    'MaxValue': 1500,
    'MinValue': 0 ,
    'DeviceId': HVL-PLC_1,
    'TagName': ST_01_X,
    'NodeId': ns=4;i=338,
    'Description': High speed alarm limit
};
```

Figure 164 Predefined information models

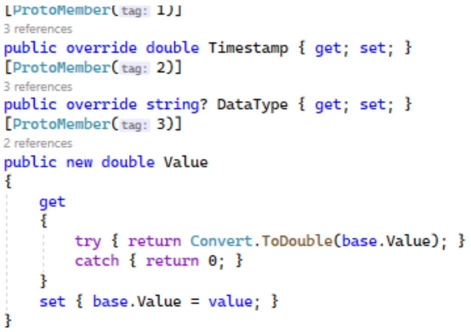
A base class is defined, which all other information models inherit from, and all models are given an enum represented by an int number to identify the InformationModelType inside the application. The base class defines attributes required by all models and required methods. The methods are not required and are transparent to the receiver at the other end. These methods strictly make the models compatible with the gateway application by guaranteeing the necessary methods to load and store models from the offline storage and move data internally inside the application. It is essential, and it has been verified that there is no added size when the models are serialized before data transfer.



```
public class InformationModelBase
{
    13 references
    public virtual double Timestamp { get; set; }
    13 references
    public virtual string? DataType { get; set; }
    13 references
    public virtual object Value { get; set; }

    7 references
    public virtual Dictionary<string, object> GetAttributes()
    {
        return new Dictionary<string, object>();
    }
    7 references
    public virtual InformationModelBase SetPublishObject(Dictionary<string, object> attributes)
    {
        return this;
    }
}
```

Figure 165 Base class for information models

The property Value is defined as type object since it must be able to take any type dictated by the deriving class. It requires some clever tricks to override the property with a different data type, but then it is possible to achieve maximum flexibility while still adhering to the base class. There is a need to include a try-catch statement when setting the value. It is essentially a measure that will never occur since the configuration of new mappings inside the application makes it impossible for the exception to occur. Still, it is good to include it for additional robustness.

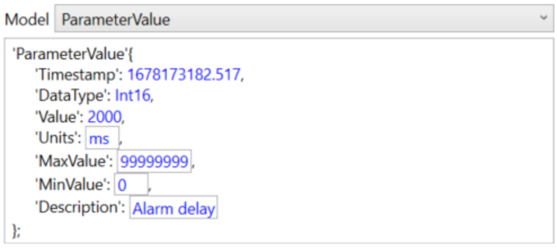


```
[ProtoMember(tag: 1)]
3 references
public override double Timestamp { get; set; }
[ProtoMember(tag: 2)]
3 references
public override string? DataType { get; set; }
[ProtoMember(tag: 3)]
2 references
public new double Value
{
    get
    {
        try { return Convert.ToDouble(base.Value); }
        catch { return 0; }
    }
    set { base.Value = value; }
}
```

Figure 166 Overriding Value property of Base class

The Protobuf-net Nuget package is included to serialize models. It puts restrictions on the properties defined in the information models. Models that are to be compatible with Protobuf are declared with the header `[ProtoContract]`. And all the properties are assigned an int identifier `[ProtoMember(tag: 1)]` used to optimally compress the data during serialization. The same models can pass data internally in the application or be serialized using Json encoding. Then these restrictions on the attributes are ignored.

Information models integrated into the application require a class that defines the structure of the model and a UI component for display to the user. Then an object that holds the ongoing runtime configuration of a new mapping must be instantiated in ConfigureItemViewModel. The object is the binding between the View and the ViewModel behind it.



Model ParameterValue

```
'ParameterValue'{
    'Timestamp': 1678173182.517,
    'DataType': Int16,
    'Value': 2000,
    'Units': ms ,
    'MaxValue': 99999999,
    'MinValue': 0 ,
    'Description': Alarm delay
};
```

Figure 167 UI component for information model

The models defined for the application are meant to be proof of concept models. Input should be collected from different stakeholders before a company commits to the system and deploys it fully. These inputs should be the basis for including attributes and choosing data types. And hopefully, it will reduce the risk of having to redesign the information. It will probably lead to fewer models overall.

11.13 TreeViewNode

The data's structuring and context are central to handling information in the unified namespace. The OPC Foundation has defined services in the specification for browsing nodes. On the OPC UA server, every node has a reference attribute pointing to the parent node and the nodes below it in the hierarchy. These references are used to dig down easily into the information hierarchy and contribute to the data context. The client will send a series of request-response operations to the server, which then responds by sending the nodes' information on one level down. These references can then be used to search even further down the hierarchy.

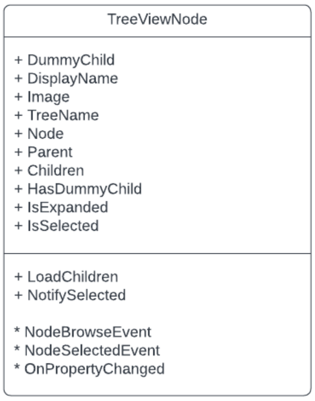


Figure 168 TreeViewNode class diagram

A WPF application is built on the principle that the UI should only be used for displaying data. All processing must take place in the code behind. It provides complete control over the data but requires a system for storage and organization in the ViewModel. A solution is to create a structure where each node is an instance of the same class and has an attribute that is a list of child objects from this same class. The list can either be empty or contain an unlimited number of children. In addition, the class has attributes to hold the information itself, which is the justification of the node`s existence, and methods to handle the further expansion of the structure.

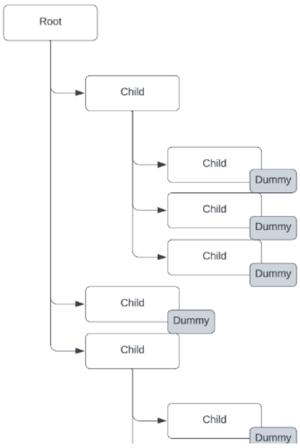


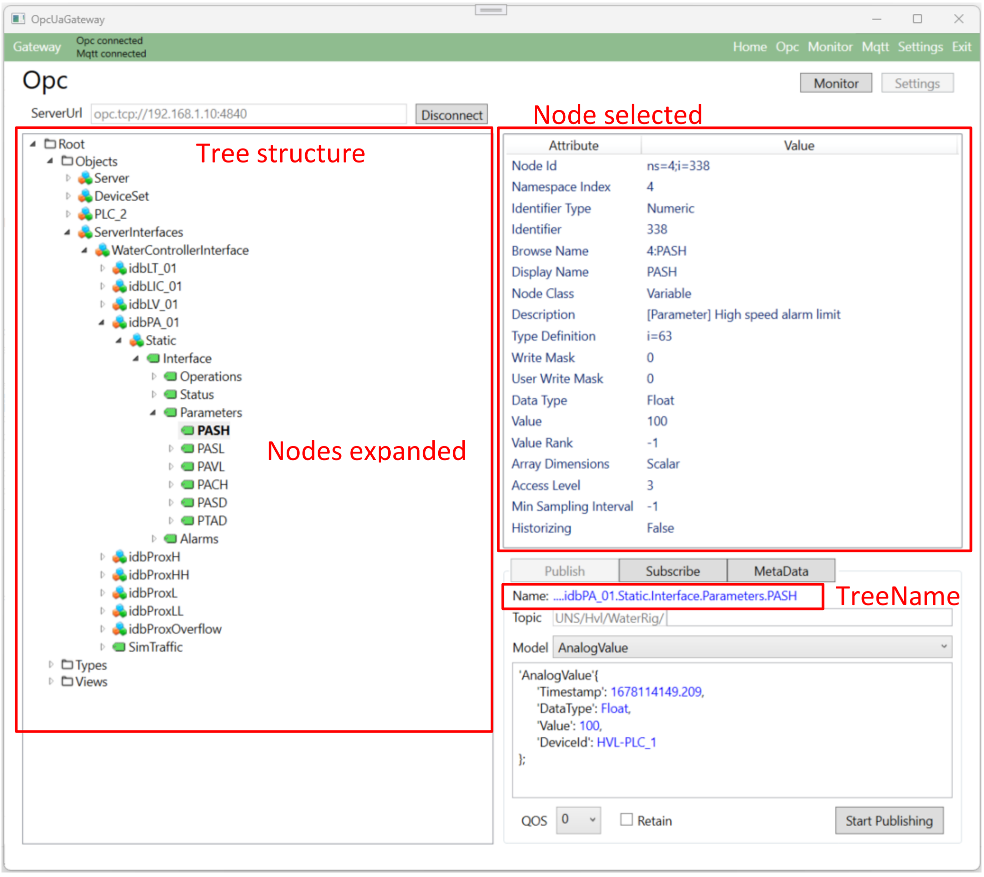
Figure 169 TreeViewNode Hierarchy

An address space on an OPC server can be significant. Browsing all of the space is unneeded and will contribute to unnecessary traffic on the network. The most sensible and what people do is only to read one node at a time and let the user manually maneuver down the hierarchy. The class has the attribute IsExpanded to handle the expansion of the tree structure. It will initially be set to false. When a node is expanded in the UI, it will send a browse request forwarded to the OPC server to get information about the nodes below it in the hierarchy. A node must have children in the list to get the symbol and the function to browse in the UI. Therefore, when instantiated, all nodes obtain a DummyChild to achieve this function and simultaneously avoid browsing the node`s children. The DummyChild is deleted when the node is further expanded.

The attribute IsSelected is used to send node read requests to the server when the node is selected in the UI. A browse request only provides information necessary to search further down the hierarchy, typically browse name and node Id. It may be of interest to see extended details of the nodes while browsing, but unconditionally retrieving it for all the nodes should be avoided. The extended information about the node is displayed on the right side of the UI.

A node will send events for browsing or reading. The code behind the view will have to subscribe to these events to forward them to the server. The reference to the sending node needs to be attached to the event handler to respond to these events. Subscribing to all the nodes in the hierarchy could cause memory leaks if not disposed of properly. Therefore, a node will attach its reference and send its requests up the hierarchy to the root node. Only the root will fire events. The method to read or browse will check whether the node has a parent node before an event is fired. If it has a parent, the request is sent to this parent, who will perform the same check. In this way, we have achieved that the ViewModel only needs to subscribe to events from the root node.

The nodes only hold references between each other. A TreeName is therefore built up consisting of five subsequent nodes to display which node is selected on the UI clearly. The name could have been longer, but five is chosen to have a name that gives enough information and is not too long so that it doesn't look good on the UI. It is after all the node Id that will uniquely identify the node.



ServerUrl opc.tcp://192.168.1.10:4840

Attribute	Value
Node Id	ns=4;i=338
Namespace Index	4
Identifier Type	Numeric
Identifier	338
Browse Name	4:PASH
Display Name	PASH
Node Class	Variable
Description	[Parameter] High speed alarm limit
Type Definition	i=63
Write Mask	0
User Write Mask	0
Data Type	Float
Value	100
Value Rank	-1
Array Dimensions	Scalar
Access Level	3
Min Sampling Interval	-1
Historizing	False

Name: ....idbPA_01.Static.Interface.Parameters.PASH

Topic UNS/Hvl/WaterRig/

Model AnalogValue

```
'AnalogValue'{
    'Timestamp': 1678114149.209,
    'DataType': Float,
    'Value': 100,
    'DeviceId': HVL-PLC_1
};
```

QOS 0 □ Retain

Figure 170 Different components of the TreeViewNode class displayed on the UI

11.14 .Net design principles

11.14.1 MVVM

The Model-View-ViewModel (MVVM) design architecture was developed to decouple the View from the code behind, allowing for benefits like development to be distributed among individuals. For example, it is possible for a designer who may not be a skilled programmer to design the visual user interface without concern for the application logic. It also makes it easier to structure the code behind a specific view, as developers can focus on the code for a single view in isolation, and variable names can be reused for different views.

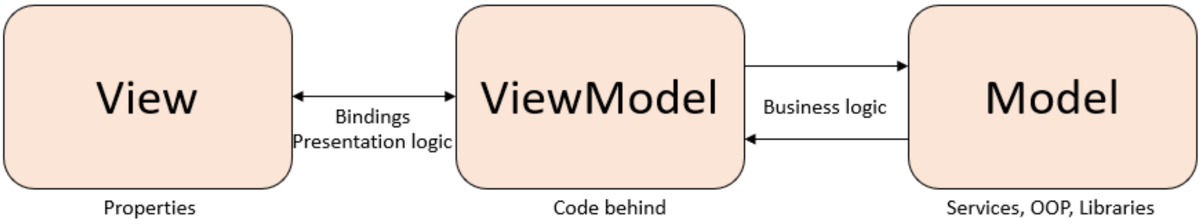


Figure 171 MVVM principle

11.14.2 Dependency Injection

In DOT NET 6.0, Dependency Injection (DI) is a standard feature. DI architecture involves instantiating objects like communication services outside the classes that will use them. Classes requiring objects from DI will receive them through the constructor. This approach reduces the coupling between classes. Additionally, some classes are implemented and registered with an interface in DI, allowing for the replacement of the underlying class with no concerns for the rest of the application logic. For example, replacing the module that communicates with the OPC interface with another module is possible as long as it implements the methods and properties defined in the interface.

11.14.3 Bindings

Bindings are the connections between objects in ViewModels and Views when using the MVVM architecture. When set up correctly, they establish a relationship between the graphical user interface (GUI) and the code behind. The DataContext which can be seen as the root address, is typically set to the ShellView in the App.xaml.cs file. All bindings to ViewModels further down the hierarchy are configured inside the ShellView.xaml file. This ensures that the different View only has access to the properties on the associated ViewModel. Proper bindings configuration is crucial for establishing seamless communication between ViewModels and Views in MVVM.

11.14.4 Event-based

The principle of only performing certain actions when an event occurs is also commonly used when programming in the .NET environment. Event-based conceptually resembles the publish-and-subscribe communication pattern used to exchange data in UNS (Unified Namespace System). In this pattern, different code modules can be programmed to wait or subscribe to events from other modules, and then execute their code when the event is triggered. It is also possible to send arguments in these events, allowing for dynamic updates such as updates to the GUI interface or arrival of new data on communication interfaces that cannot be predicted in advance. This event-driven approach provides flexibility and modularity in the software design, allowing different modules to communicate and react to changes or events as needed.

11.14.5 Asynchronous programming

Tasks can require varying amounts of processing and occur at unpredictable times in a program that actively interacts with its surroundings. Asynchronous programming is a technique that prevents tasks from getting in each other's way. For example, it can effectively prevent modules waiting for a response to a network request from causing others to wait. It can also be useful for computationally heavy tasks where overall processing can be improved by spawning the task on a new thread.

11.14.6 Stores

Some variables need to be accessible anywhere in the program. These variables or objects are created as "stores" registered as Singletons in DI (Dependency Injection) and injected via constructors into the classes that require them.

11.14.7 Services

Service-oriented programming (SOP) is a design approach focusing on defining tasks or services that other parts of an application can request. These services are encapsulated as separate modules or components that can be invoked or utilized by other parts of the application without knowing the implementation details. An example of this is a navigation service, where the user can request to change the current window by requesting the new window as an argument to the navigation service.

Service orientation is particularly useful in scenarios where different parts of a system may be written in different programming languages, or when there is a need for loose coupling between components to enable flexibility. By encapsulating functionalities as services with well-defined interfaces, the focus can be shifted from implementation details to how services are processed, allowing for easier integration and interoperability in complex software systems.

11.14.8 Interfaces

Interfaces can be seen as a contract that all classes that implement the interface must adhere to. The advantage is that other classes that interact with this class know that it has a minimum level of functionality through the implemented interface. Interfaces make it easier to replace classes or change the class's internal code without affecting other parts of the application.

11.14.9 App settings

Settings inside the application are stored in the 'appsettings.json' file, a common location that can be accessed via the IConfiguration object. In DOT NET 6.0, IConfiguration is included in Dependency Injection.