Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Einar Marstrander Omang

OPC-UA Interface for Safety Instrumented Systems

Master's thesis in Cybernetics and Robotics
Supervisor: Mary Ann Lundteigen
Co-supervisor: Arvid Bjarne Nilsen
May 2021

NTNU
Kunnskap for en bedre verden

Einar Marstrander Omang

OPC-UA Interface for Safety Instrumented Systems

Master's thesis in Cybernetics and Robotics
Supervisor: Mary Ann Lundteigen
Co-supervisor: Arvid Bjarne Nilsen
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Kunnskap for en bedre verden

Preface

This is a master thesis concluding a master's degree in cybernetics and robotics at NTNU. It can be considered a continuation of the autumn 2020 specialization project[34] "APOS OPC-UA" written for SINTEF autumn 2020. At NTNU, a half-semester specialization project is completed before starting the single semester master thesis, and the two are frequently related.

The specialization project, and by extension this thesis, build on the APOS (Automatisert prosess for oppfølging av instrumenterte sikkerhetssystemer, English: Automated process for follow-up of safety instrumented systems) project at SINTEF. APOS, among other things, suggests an information model for equipment and failure classification in safety instrumented systems. OPC-UA is a standard for industrial communication which includes both a platform for communication, and a generic information model. The specialization project designed an OPC-UA implementation of the APOS information model, and the goal of this project is to explore the challenges and limitations of applying such a model to a practical case at Aker BP. This thesis details the development of such an OPC-UA server.

In practice, this requires obtaining access to Aker BP information management systems, relating the contents to the APOS information model, and creating an OPC-UA server that exposes this information automatically. Aker BP source systems are for the most part not designed to be accessed from external applications, and so in order to find ways to access the information, a large part of this project involves reverse engineering the source systems.

Because of this, the thesis touches on a wide variety of technical areas. The theoretical foundation is presented as part of this thesis, but the specialization project goes into further detail. In general, the project assumes a general technical background. The basis for this project is safety instrumented systems and OPC-UA like the specialization project, but also the world wide web, modern web development, database management, and web APIs.

Using the data specific to Aker BP systems in described in this text in a commercial setting requires permission from Aker BP. The OPC-UA server and information model described in this project is not in use in any Aker BP systems.

This project was completed with support from Mary Ann Lundteigen (NTNU) as primary supervisor, and Arvid Bjarne Nilsen (Aker BP) as secondary supervisor. Thanks to everyone in Aker BP who has provided support: Erik Stangborli for help with the LCI database, Kristoffer Lanne, Anders Nystøl, Kristian Førland Steinsland and Bjørn Hauge Hansen for help with SAP, Lars Søraas (Sharecat) for help with getting access to the EqHub API, and Knut Omang for reviewing the thesis. Thanks in general to Sharecat who put in time and effort setting up suitable access to their API. A big thanks also to Aker BP IT-support for help with getting access to the various systems, as well as with getting in contact with the relevant people in Aker BP and elsewhere.

Oslo, 28/05/2021

Einar Marstrander Omang

Executive Summary

This thesis explores the challenges and limitations of creating an OPC-UA (Open Platform Communications - Unified Architecture) server based on the OPC-UA information model for APOS developed in the autumn 2020 specialization project APOS OPC-UA[34]. The APOS[39] project at SINTEF defines a standard information model for safety instrumented systems, and the specialization project suggests how this can be realized in OPC-UA. Safety instrumented systems refer to instrumented systems in industrial settings that are exclusively used to prevent and detect dangerous events. OPC-UA is a modern standard for industrial communication and information modeling.

In the field of safety instrumented systems it is important to contextualize and organize failure events in order to measure the failure rates of various field equipment types. The APOS model is developed with this in mind. Today, collecting this information involves a great deal of manual work, and so a part of APOS is also designing algorithms and standard procedures using the developed information model to make this process more automatic.

The specialization project suggests an OPC-UA implementation of the APOS model. This thesis is an extension of that project, and details the development of an OPC-UA server using the information model exposing industrial information from three information management systems at Aker BP: an LCI (Life Cycle Information) database running MS SQL Server, a SAP[17] (Systems, Applications, and Products in Data Processing) system, and a central repository for equipment named EqHub[12], managed by Sharecat[18].

Through this process, limitations and challenges in both the APOS project and the Aker BP source systems are identified. First, gaining automated access to the source systems is difficult, as they are in some cases not designed to be accessed in such a way. Secondly, fitting the data in the source systems to the APOS information model was generally difficult, and in some cases impossible due to inconsistent or missing data in the source systems. Finally, once access was obtained, accessing the data was slow enough that some applications of the server became impractical.

Based on these challenges, concrete recommendations for the APOS project and Aker BP are suggested. The APOS project only applies to safety instrumented systems, which may make it more difficult to adapt in systems that structurally do not differentiate between safety functions and normal equipment; Both SAP and LCI are slow and difficult to access automatically; All three source systems have trouble with consistency, which makes automatic mapping to the OPC-UA model difficult; Finally, the information model is in some edge-cases potentially incompatible with the ISA-95 standard.

Contents

Chapter 1

Introduction

1.1 Background

An interface is defined as the environment by which two processes interact. In information technology (IT), this is an essential term. A modern computer consists of an enormous number of separate systems, developed by numerous organizations and people, and often involve decades of work. Understanding every piece of a system like this is not feasible, and so in order to work with it, it must be simplified. Interfaces are at the core of this simplification. By defining limited, well-defined interfaces between components, new processes can be attached to the existing system without requiring a full understanding of the underlying processes behind the interface.

The OPC-UA standard is such an interface. It defines a platform for industrial communication, which includes a flexible information model. It does not define the implementation of any server or client using the OPC-UA standard, but it does provide sufficient details to facilitate the creation of consistent OPC-UA interfaces for different systems. By applying an OPC-UA information model to an existing system, the complexities of the underlying system is abstracted away, and the data can be accessed with only an understanding of OPC-UA.

While the OPC-UA information model is rich and flexible, it is by design not specific to a specific industry, and it is intended to be expanded for use in particular settings, using the companion standards, or other OPC-UA based information models. The *specialization project* that this thesis is based on, "APOS OPC-UA"[34], describes an OPC-UA information model for the APOS project[39] (Automatisert prosess for oppfølging av instrumenterte sikkerhetssystemer, English: Automated process for follow-up of safety instrumented systems). Safety instrumented systems refer to systems that are exclusively used to prevent and detect dangerous events, and as such have strict requirements for follow-up, to guarantee that the systems provide adequate protection. APOS defines an information model, and procedures based on this model, for partially or fully automating this follow-up process.

OPC-UA was chosen because it is flexible enough to contain the APOS information model, and because it is widely used in industry, especially in Europe. An OPC-UA based information model has the advantage of being readable by users not familiar with the specifics of APOS, while being flexible enough to contain the desired level of detail.

The specialization project is able to create a generic mapping, and explore how the different requirements of APOS are achievable in OPC-UA. While the model should in theory be able to

completely encapsulate the APOS model, it is not complete, as in order to limit the scope of the specialization project, it only implements a small part of the APOS information model. Instead of attempting to expand the model to cover the entire APOS hierarchy, which would likely be a long and difficult, manual process, this project aims to further develop and verify the model by applying it to an existing industrial information management system at Aker BP.

Aker BP is an APOS partner, and has agreed to give access to their information management systems. Aker BP is a major oil company in Europe, focused on discovery and extraction of oil in the north sea. It is also notably a result of a 2016 merger of BP Norway, Det norske oljeselskap (DETNOR, The Norwegian oil company), driven by BP and Aker. This also has the consequence that one of Aker BPs platforms was operated by the American oil company Marathon Oil until 2014, and another was operated by Amoco until it was acquired by BP in 1999, so they manage platforms that were originally developed in at least four different companies, which ages range from 4 to 40 years.

This has some consequences for the technical systems in use, as Aker BP is in a process of merging the various systems used by the companies that originally operated each platform. Most relevant to this project, they are in the process of creating a unified OPC-UA hub, and are looking to APOS for a possible information model.

Much of the IT work in Aker BP has been performed by third parties, which means that obtaining information about the technical systems in use can be challenging. The consequence of this is that due to time limitation and lacking or unavailable documentation, it is necessary to obtain information about the source systems without external help. Fully understanding this work requires a fundamental understanding of the underlying protocols of the world wide web, and how modern websites are built. The various terms will be explained as they come up, but understanding the background for the steps taken to explore the data may take some further reading.

There exists a number of relatively simple sources for understanding the core concepts. “Understanding the World Wide Web: A Brief Primer” by Courtney Hunt[31] provides a very brief introduction to the key concepts, but importantly also points to other sources for further reading.

The source information for any literature about the world wide web is going to be the original standards and documents. There are numerous publications that lay the foundation for the modern world wide web, but the key ones are the standards published by the “World Wide Web Consortium” (W3C)[22]. W3C publishes a number of standards for the modern web. The pages on JavaScript[15] and HTML[14] are particularly relevant for this project.

The primary sources for the core of the internet are the RFCs (Request For Comment) published by ISOC (Internet Society), primarily IETF (Internet Engineering Task Force). Of particular note here is RFC 2616[27] on HTTP 1.1 and RFC 6749[30] on OAUTH 2.0.

The background for the APOS project is mostly covered by the specialization project, and only briefly here. The book “Reliability of Safety-Critical Systems. Theory and Applications”[36] by Marvind Rausand is used as theoretical basis for safety instrumented systems. The source material about APOS is primarily the H1 report[37], which contains the foundation for the information model, and the H5 report[38], which was used to create the information model in the specialization project. The APOS project is itself based on a variety of IEC standards, in particular IEC-61508[4] and 61511[25], as well as ISO-14224[29].

Beyond that the work is based on the official OPC-UA standard, which can be found online[35], and the sample applications[19] for the OPC-UA SDK[20] created by OPC Foundation. Some in-

formation on the internal systems have been obtained from internal sources in Aker BP, and the rest has been obtained from publicly available documentation, or by reverse engineering the source systems.

1.2 Objective

The goal of this project is to identify limitations in the APOS model, in the OPC-UA model for APOS, and in a few of the information management systems at Aker BP, and through this make concrete recommendations to APOS and Aker BP.

In order to achieve this, the master project implements a functional OPC-UA server that exposes real industrial data from Aker BP through the APOS OPC-UA model developed in the specialization project. The thesis explores the problem, and through this attempts to identify what needs to change in both the source systems and the information model in order to create a solution using the APOS information model, that could be included as a part of a live, industrial information management system.

The following three key questions are central to the thesis:

1) What requirements should be posed to APOS to make it possible to develop practical OPC-UA servers using the APOS information model?

2) Is the model developed in the specialization project sufficient to model the data found in Aker BPs systems?

3) What requirements should be posed to the contents and technical implementations of information management systems in order to make them compatible with the type of OPC-UA server described in this project?

Some of these questions are explored in the specialization project, but without any testing, any conclusions end up being speculation. Similarly, this project is also limited in scope, since it only explores data in Aker BP. In order to make conclusions about the state of the industry in general, other companies would need to be studied in further detail. Still, other companies, especially in the same field, may have similar challenges, and the general requirements for the OPC-UA server would also apply to a different set of information management systems.

1.3 Approach

The majority of the theoretical basis for this task has already been explored in the specialization project, so most of this project is work related to the practical implementation and the issues this exposes. The text lays the foundation for the server implementation, then proceeds to describe the creation of a working OPC-UA server.

The development of the server can be divided into four general stages, this is the first half of the master project:

1) Understand how such an OPC-UA server should be written.

2) Identify the source systems in use at Aker BP.

3) Create a procedure to map source system data to the APOS model.

4) Implement an OPC-UA server presenting the source system data in the APOS information model.

Item 1) requires a fundamental understanding of the OPC-UA standard and the APOS information model developed in the specialization project. Most of this is covered in detail in the attached project report, and so this thesis only briefly summarizes the key points. In order to properly understand the development of an OPC-UA server, a solid understanding of the interface itself is also required, which in OPC-UA means discussing the *services*.

The actual server implementation uses the official OPC Foundation OPC-UA .NET SDK (Software Development Kit)[20]. The server created as part of the specialization project was simple, with almost no code that could not be found in the sample repository. For this project, however, OPC-UA services must be implemented to read from underlying systems, which requires considerably more work and a deeper understanding of the OPC-UA standard.

Item 2) means making an effort to understand the source systems. How are they used, how do they work, and how can they can be accessed by an external computer program. In order to ensure the completion of the server, it was decided to reverse engineer the source systems, in order to discover a way to access the information without relying on APIs (Application Programming Interface) provided by Aker BP.

Using the understanding of OPC-UA from 1) and the source system data from 2), 3) can be completed by creating a procedure to map data from the source systems into the information model developed by the specialization project. Finally, using this, 4) combines the methods to access the source systems, the relevant OPC-UA services, and the procedures for information mapping to create a functional OPC-UA server providing access to the source systems.

The second half of the thesis uses the server to answer the questions posed in section 1.2 by first testing that the server is capable of handling the APOS model and the source system data, by expanding the server to cover more types of equipment, then attempting to connect to the server and use the data. Using the results from these experiments, it is possible to answer all three questions.

1.4 Outline

The thesis is effectively divided into two parts. Chapters 2 to 6 implement the server, and explore the concrete challenges related to this. The second half, chapters 7 to 9 evaluate the server and explores the potential limitations of the source systems and the APOS project itself.

Chapter 2 discusses the information model laid out in the specialization project. Using this as a basis, chapter 3 describes the various OPC-UA services needed for an implementation of this model, and what it means to build an OPC-UA server on top of an underlying system. This chapter also describes what kind of information the various services will need from the source systems. How to extract this data is explored in chapter 4, which identifies the APIs available in the three information management systems needed for the implementation of the APOS model.

In chapter 5, the source system information models are translated to the APOS model, and in chapter 6 this is used to create a functional server implementation.

In the second half of the thesis, chapter 7 populates the model with data for two equipment groups, and uses this to discuss how difficult it might be to expand the server to cover the full APOS information model. Chapter 8 studies different practical uses of the server, by connecting to it with three external applications. Finally, chapter 9 attempts to answer the three questions above using the results from the rest of the thesis.

Chapter 2

Background

When organizing data it is useful to structure it in an information model, which typically consists of a list of information types, constraints for each type, and rules for how the different types interact. Here, the information model is developed as part of the specialization project[34]. This chapter briefly summarizes the theoretical basis for the APOS project, the APOS project itself, and the information model developed for the specialization project.

The specialization project goes into further detail on each topic discussed in this chapter, and can be used as a source of further information. In particular, chapter 2 concerns safety instrumented systems, chapter 3 discusses core concepts of OPC-UA, and the APOS project is covered in chapter 4. Chapters 5, 6 and 7 establish the information model, and discuss how it could be expanded.

2.1 Safety Instrumented Systems

A SIS (Safety Instrumented System) is a safety system that uses active instrumentation. This is unlike passive safety systems, which have no active instrumentation. A SIS typically consists of a number of SIFs (Safety Instrumented Function), that each alleviate a specific danger.

The APOS project uses some terminology specific to this field, which the reader should be familiar with. These definitions are as described by the APOS H1 report[37], which is based on general IEC terminology[26], and IEC 61508[4]:

Failure refers to the inability of a system to perform its intended task. A **fault** is an underlying issue, which can remain undetected for long periods of time, but then cause a failure due to demand or unusual conditions.

Failure Mode is the way a failure affects the system, describing the specific way a component was unable to complete its intended task.

Failure Detection refers to the way the failure was first observed, either by an automatic system, or by an operator.

Failure Cause is one of the causes of a failure. This is typically one or more faults, combined with unusual conditions.

Failure Class refers to a division into four classes, classifying events as **Dangerous** or **Safe**, and **Detected** or **Undetected**. For safety analysis, only **Dangerous Undetected** (DU) failures are relevant, as it is assumed that all others cause the system to shut down, or somehow make further danger impossible.

The goal of most safety analysis is to determine the root cause of a failure, to make changes that make the failure less likely in the future, or ensure that it does not cause a dangerous event that could further harm people or property.

2.2 APOS

The APOS project is a project at SINTEF that aims to develop specifications for automating and organizing work with safety instrumented systems primarily in the petroleum industry. The focus of this project is on the information modeling part of APOS, which has two main components[37]:

The equipment hierarchy defines a system for grouping of equipment, based on two main criteria: The *function* and *design* of each piece of equipment. It has three levels, designated L1-L3.

L1 groups equipment by its main purpose, i.e. gas detection, fire detection, shutdown valves, etc. L2 groups group equipment by core operating principle or design. For example, the gas detector hierarchy divides based on measuring principle (line, point, etc.) and measured gas (Hydrocarbon, CO, H2S, etc.). The L3 level contains a collection of attributes for each specific equipment design, with further details such as intended location, available self-diagnostics, exact measuring principle, etc.

APOS also proposes hierarchies for failure modes, causes and detections, defining D1-D2 for detections, F1-F2 for modes, and C1-C2 for causes. The purpose is to help classify failures using a hierarchical model with increasing specificity descending the tree. An automated system might be able to select which F1/C1 level a failure belongs to, but need operator input to select F2. There may be common responses to C1 failure causes that do not need to be specified for each C2 cause.

2.3 Model

The specialization project[34] chapter 5 creates an abstract model independent of OPC-UA for the APOS model, and chapter 6 adapts this model to OPC-UA. This is seen in figure 2.1, which is a slightly modified version of figure 6.15 in the specialization project report. It contains four core types of information, separated by color. Each of these represent different types of information in the source systems.

Instance Information

The green objects refer to *instance information*, meaning that they represent physical or inferred properties of real or logical objects. They always refer to something concrete, attached to a single

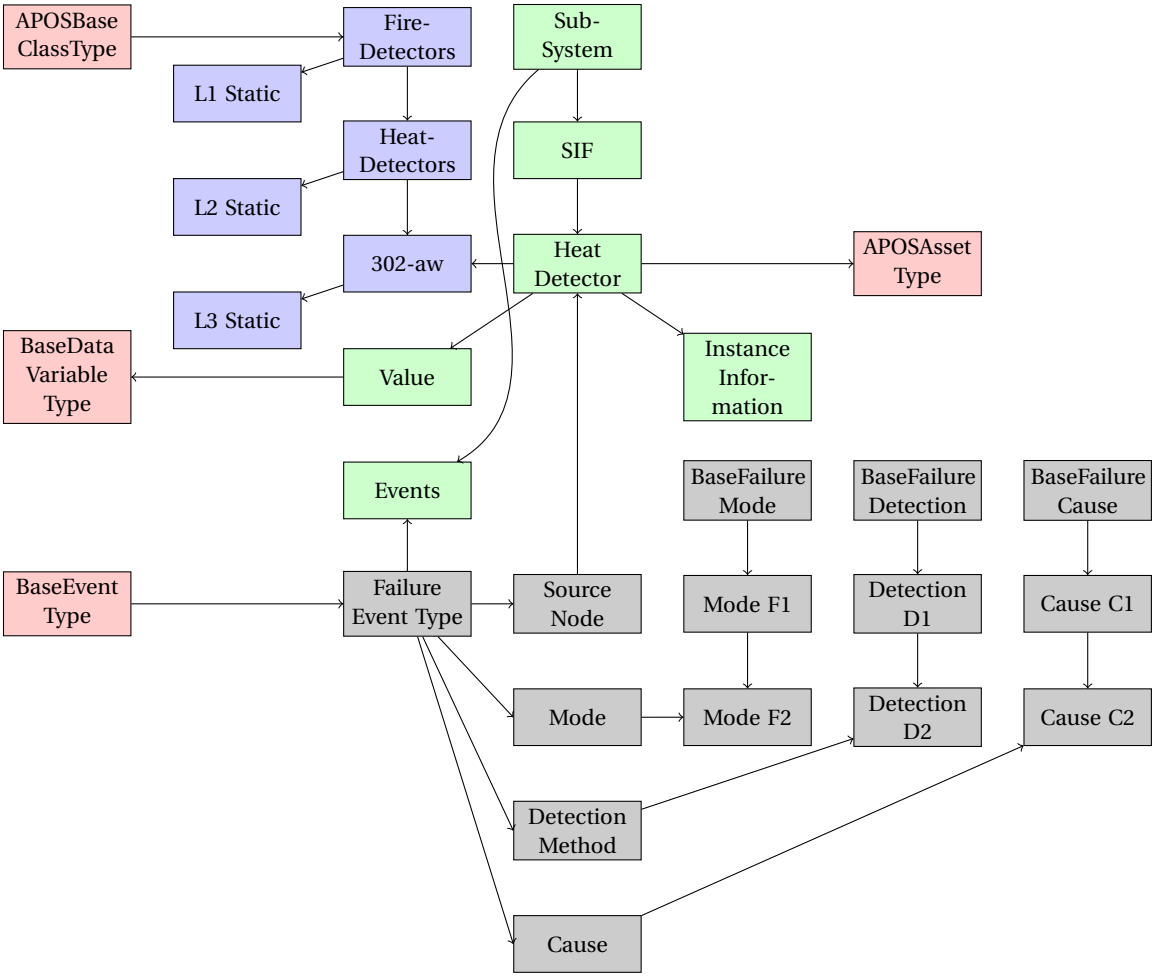


Figure 2.1: Implemented information model.

instance. Figure 2.1 shows a single heat detector, which has properties like location and age, a live measurement, and a collection of sporadic events that may represent past failures.

When reading from source systems, instance information is generally the easiest to identify. Any information that refers to a specific instrument or physical asset is placed in this group. Typically, this will mean that if information is not better represented as any of the three other types, then it must be instance information, so if information does not refer to all equipment of the same model, or does not describe sporadic events, then it must be instance information in some way.

Equipment Type Information

The blue objects refer to the APOS equipment hierarchies. These are divided into three levels, here exemplified by “Fire Detectors” as the L1 level, “Heat Detectors” as the L2 level, and “302-aw”, a specific heat detector model, as the L3 level. Equipment instances are connected to L3 layers, as a way to identify the make and model of each physical equipment. Each level has static type information, which gets increasingly specific deeper down in the hierarchy. L1 static information only describes the possible attributes for each type, and perhaps some common

properties, L2 is similar, but slightly more specific, and L3 reflects the specific properties of each equipment model.

When retrieving this information from source systems, there are two issues. First, each instrument should belong to an L3 type. This type needs to be identified, and it must be assigned to a L2 category. Optionally, it might have to be placed directly in an L1 group if there is no suitable L2 group.

Ideally, multiple pieces of equipment will belong to the same L3 type, however there is no guarantee that the source systems use such a classification. It might be possible to group equipment using other information using something like model name, model number, etc. In that case, the static type information would have to be extracted from each type, and the server would have to find some consensus if there are conflicts between different types.

The model also suggests that there should be strict data types for each value, but this is limited by the information model used by the source system. If there are no type limitations in the source system, defining data types may not be practical. It might be possible to create a few predefined types, then mapping the values in the source system to these, however that runs the risk of a value in the source system being impossible to report.

Event Information

The gray objects describe failure events, and classify them using the three core concepts in safety instrumented systems: failure mode; how the failure happened, failure detection; how it was discovered, and failure cause; why it happened. The information mapped to events is typically logs of failed tests or triggered failure notifications.

2.4 OPC-UA Mapping

Translating this abstract model to OPC-UA means deciding on how each type of information can be represented in OPC-UA. OPC-UA is a node hierarchy, a collection of nodes with references between them. The first step in creating the mapping is deciding on the *NodeClasses* for each type of information, four NodeClasses are most relevant to this project. These and the remaining four are discussed in depth in section 3.3 of the specialization project.

In general, the contents of this section is just a summary of the much more detailed discussion in the specialization project report chapter 3, which also contains information about communication in OPC-UA, further details about the basic information model, and more information about the ISA-95 companion standard, which is a commonly used extension to OPC-UA for modeling in industrial systems.

An **Object** represents a physical or logical concept, objects are used to connect the other node classes together. They are shown as circles in figure 2.2. They typically represent anything that does not fit into any of the other classes, and they contextualize the data contained in other nodes.

An **ObjectType** or **VariableType** is used for describing objects and variables in some way. Variable types are almost exclusively used as templates for variables, while object types are also used for nodes in the type hierarchy. Only one variable type is shown in figure 2.2, “BaseDataVariableType”, which represents a variable that contains changing data values. The other

important one is "PropertyType", which indicates that a variable contains largely static values. Both ObjectTypes and VariableTypes are shown as rectangles with thick edges.

Object types are used for three main purposes here. First, they define the "template" for a node, this is the base OPC-UA interpretation of the object type: each type has a "TypeDefinition", which describes what the node represents, and may specify what kind of properties and values it should have. Secondly, they are used for event types. Events have a finite set of properties, which must be described in detail in the type hierarchy. Finally, they are used by ISA-95 to describe *class types* for nodes.

The ISA-95 companion standard defines four different class types, two of which are relevant for this project. They differ from type definitions in that they represent common properties for a node, instead of a template. So multiple nodes can share a class type, and the properties of that class type will effectively apply to all associated nodes, which avoid information duplication.

The first of the two relevant class types are "Physical Assets", which represent concrete physical objects like a specific kind of detector, a specific brand of vehicle, or similar. Physical asset classes describe common properties of multiple physical assets, and each physical asset can only have one physical asset class. For example, the class could be a specific model of gas detector, and each physical asset which uses that class would represent an instance of that model in the field.

The second class type is *Equipment*, which refers to a more abstract idea of "role" for each node. For example a gas detector might have both the "gas detector" role, and the "safety critical equipment" role. An Equipment node will typically consist of a collection of other equipment and physical assets. This distinction between Equipment and Physical Assets was misunderstood in the specialization project report.

The original model only uses physical assets, the L1, L2 and L3 equipment groups are physical asset class types, and the "APOSAssetType" is a sub type of the "PhysicalAssetType", which is a normal OPC-UA object type. This means that here the "302-aw" node is a specific model of heat detector, which contains common information shared between all instances of that model.

A **Variable** is simply a node that contains a value. Variables are represented by both rectangles with thin edges and diamonds in figure 2.2. The difference between the two is their Type Definition. Rectangles with thin edges are data variables, these often reflect some measurement or similar dynamic value. Diamonds are properties, meaning that they represent static information.

Finally, note the "Events" node which does not have a type at all. This is because events in OPC-UA are not represented as nodes. In fact, OPC-UA does not describe how events should be represented internally at all, only how they should be displayed to the user. This is discussed further in chapter 3.

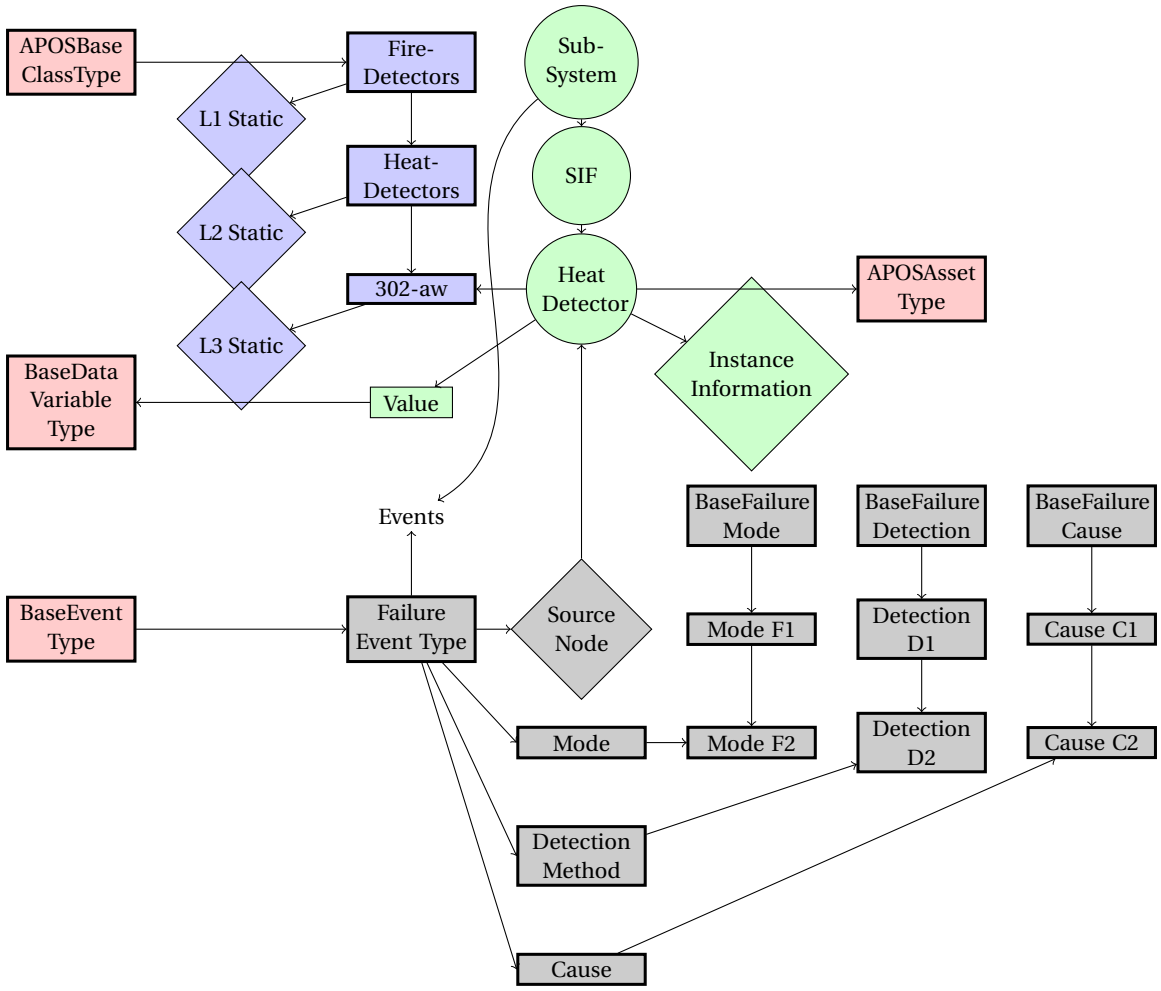


Figure 2.2: Information model in OPC-UA.

Chapter 3

OPC-UA Services

Three parts of the core OPC-UA standard: information modeling, data encoding and security are described in chapter 3 of the specialization project. The next part of OPC-UA is its API, discussed in part 4 of the standard[35]. The API is the interface an OPC-UA server exposes to the world. This comes as a collection of around 40 "Services", which each provide some way of interacting with the data exposed by the OPC-UA server. In order to develop a server that provides data using the APOS information model, it is necessary to understand what some of these services do.

When developing an OPC-UA server it will always, by necessity, expose some underlying system. The OPC-UA standard does not define how information is to be stored, only how it should be presented. This means that it is up to the developer to decide on a reasonable mapping not just for the information model, but also for how requests to the interface should be translated. Even the server in the specialization project was based on an underlying system, even if it was just an in-memory node hierarchy on the same computer.

For example, an OPC-UA server running on some field PLC (Programmable Logic Controller) might work like this: It will have a predefined and somewhat configurable structure which is stored in its permanent memory, if a user wants to read this structure, the OPC-UA server will translate the generic OPC-UA service calls into commands to access the relevant parts of its memory. It may also have some sensor values that may be stored in some more volatile memory, calls to read these values will be translated to the correct form of memory access, and so on.

This type of in-memory server is simple, but very common. The server developed for the project thesis had its structure stored entirely in permanent memory (the NodeSet file) or in code. A more complex system might have an OPC-UA server running on some more powerful infrastructure server, providing without replicating the data from a number of field PLCs that contain the actual information. This means that a service call to read information about some device defined on the OPC-UA server may be translated into network requests to obtain information from the PLCs.

This is one of the strengths of OPC-UA, it allows users to ignore the underlying structure, and access information from several underlying systems using the same, limited collection of services. In theory, you could write an OPC-UA server where the computer just prints service calls on a screen, and an operator types in the correct response (although this would be a very slow server).

There are several services related to underlying communication and session-management.

While these are needed if one wanted to develop an OPC-UA server from scratch, this project will use an SDK (Software Development Kit) that handles a number of services automatically. An SDK is generally a collection of software used to simplify development by providing a higher level interface to some system or concept. In this case it makes it possible for this project to ignore the low-level concepts in OPC-UA, like session logic, data serialization, and TCP. Instead, this chapter will cover only the services necessary for creating a server using the APOS model, that gets its information from external systems.

3.1 NodeIds

In order to understand the services it is important to understand a core OPC-UA, the *NodeId*. The idea of a globally unique identifier is not something unique to OPC-UA. In fact, it turns out that defining a unique identifier for each piece of information greatly simplifies design and use of information models and information storage systems. That in itself is not so strange, it is in fact difficult to imagine some system without a unique identifier at all, since unique identifiers can be made arbitrarily complex. A node in the basic information model developed in the specialization project report[34] could, for example, be defined uniquely by its position in the node hierarchy.

A model without a unique identifier at all would potentially have different items that are impossible to distinguish, which is not very useful. If two items are completely indistinguishable, then they are the same, and it would in fact be impossible to store them in any reasonable way, since the act of, for example, assigning identical entries indices, would expand the model to effectively give them a unique identifier.

That said, there are pure programmatic benefits of having strictly defined unique ids. It can make lookup more efficient, and importantly it makes it possible to carry a reference to a piece of information and pass it along to other systems. In the case of OPC-UA, this means that the server can give the client a compact NodeId, instead of a more complex identifier, which simplifies the interface.

Most systems decide on some simple system of either numeric or character sequence (string) identifiers, but as OPC-UA is not an implementation but a generic interface, it would not be ideal to make such a restriction. For example, if OPC-UA had decided that all identifiers were to be integers, and it was then used to expose some underlying system which used string identifiers, it would be difficult to create a consistent mapping, and it would almost certainly be necessary to use some kind of internal state to manage the mapping, which makes the server less “flat”.

The “flatness” is an important design criteria when building interfaces. Ideally, a call to an OPC-UA service should only require the server to do a minimal amount of work. This is achieved by ensuring that the NodeId contains sufficient information to uniquely identify the piece of data in the source systems that it references, using minimal external context.

Because of this ideal, the OPC-UA standard states that a NodeId should be able to be defined using one of four NodeId Identifier types:

- Numeric, a 32 bit integer.
- String, a string of up to 4096 characters.
- Guid, (Globally Unique Identifier), formally defined by RFC 4122[32].

- Opaque, a byte-string of up to 4096 bytes. Similar to string, but the format is even more free. An opaque identifier could easily encode *any* piece of information of less than 4096 bytes, meaning a NodeId could even be something like an image or a short audio clip.

These are commonly described on the following forms: “i=123”, “s=somestringid”, “g=5ec1d795-9b9c-4c1c-a6a6-515cc6ae53d0”, “o=QSB2ZXJ5IG5pY2UgSUQ=”, that is, the identifier type is symbolized by a single letter: “i” for numeric, “s” for string, “g” for GUID and “o” for opaque, followed by “=”, and a string representation of the identifier. For byte-string this is base-64 encoded. This text will use this format to describe NodeIds when necessary.

This, however, may still cause some inconvenient issues. For instance if one wanted to create an OPC-UA server that exposes several databases that use 32 bit numeric identifiers. Each database can use the full range of the numeric NodeIds, so it is not possible to uniquely identify a value that may come from either of these servers using just a number. One of the other identifier types could be used, but this is also not ideal, since it would be far more elegant to just use the IDs from the source systems directly. It is better to perform a minimal amount of translation, again to minimize the complexity of the interface.

This is where the idea of Namespaces is useful. In addition to the identifier, each NodeId is associated with a namespace, typically a URI[21] (Uniform Resource Identifier). For example, the base namespace for all OPC-UA servers is “http://opcfoundation.org/UA/”. In the NodeId itself this is encoded as a *NamspaceIndex*, meaning the index in the namespace-table stored on the server.

This solves the issue with duplicated IDs across databases. Now, each database is given its own namespace, and use the 32 bit IDs directly. So there can be multiple NodeIds “i=1”, so long as they each have a different namespace. This is usually written like “ns=1;i=1”, “ns=2;i=1”. If the namespaceIndex is 0, meaning it belongs to the base OPC-UA namespace, “ns=0;” is usually omitted. The other namespaces are decided by each server implementation, but namespace 0 is always the base namespace.

For a well-defined server exposing multiple underlying systems, this is how it is usually done: Define a namespace for each underlying system, and use an as close approximation of the IDs used by that system as possible. Ideally, a NodeId received from the client should be able to be immediately used to look up information about that node in an underlying system, without having to refer to some other underlying system like an in-memory table or similar.

The Namespace is chosen to represent the source system in some way. For a server that exposes a list of PLCs, the namespace of each PLC might just be its IP address. If that is not possible, then the only requirements to the namespace is that it is a URI, and that it uniquely and unambiguously identifies the source system.

3.2 Browse

There are only two fundamental services needed to read the node hierarchy. The first of these is *Browse*, which asks the server to respond with the references to and from a given node. The structure of a browse request is fairly large, here the description is limited to the parts that are most relevant for this project. A basic element of a request is the *BrowseDescription* seen in table 3.1.

Name	Type	Description
nodeId	NodeId	The NodeId of the node to browse.
browseDirection	Enum BrowseDirection	One of “Forward”, “Inverse” or “Both”. The direction of references to return, where “Inverse” means references pointing *to* the node in question.
referenceTypeId	NodeId	The NodeId of a reference type used to filter results further. This can be left out to return all types.
includeSubtypes	Boolean	A boolean value indicating whether to return references that are subtypes of the given referenceTypeId.
nodeClassMask	Integer	A mask for filtering which NodeClasses to return. This can be left as 0 to return all classes.
resultMask	Integer	A mask for filtering result fields, six different fields can be requested for each referenced node: referenceType, isForward (true/-false for whether the reference is forward or inverse), NodeClass, BrowseName (a unique name within the children of the given node), DisplayName (the human-readable name of the node) and TypeDefinition (The objectType/-variableType of this node).

Table 3.1: The BrowseDescription structure in OPC-UA.

A call to the Browse service is a list of BrowseDescriptions, as well as a limit for the max number of references to return for each. The result is a list of the returned references, and potentially a list of *ContinuationPoints*. The idea of ContinuationPoints is important, as it allows the server to return only part of the result. Say the server is asked for 100 children of a node with 1000, the server will only return 100, but it will also return a ContinuationPoint, making it possible to make another request including that ContinuationPoint to continue reading from where the server stopped instead of just getting the first 100 again.

This helps keep message sizes small. If a user instead requested children of 100 such nodes, that each had 10000 children, they might end up receiving a million nodes. If this was sent over a slow or unstable connection, the chances that it would never arrive, or that it would arrive incomplete rises. It is better for both client and server if the chunks are kept relatively small, and instead sent over a larger number of requests.

The Browse service is often the service that accesses the largest number of different systems. In the previous example of a server connecting to different PLCs, browsing a single node may return references that point to multiple PLCs, and information about each reference would need to be retrieved from those PLCs. So browsing a single node could result in requests to multiple source systems. For the client, this is very convenient, as they essentially only provided the server with a single NodeId, but actually obtaining that information involves complex operations and knowledge of the underlying systems. If the OPC-UA server did not exist, it would be much more difficult for the user to obtain this information efficiently.

Using the Browse service is often among the first things a user will do when accessing a server. Before the user can ask for the specific information they want, they need to know what nodes exist, and what their NodeIds are.

3.3 Read

The second essential service is *Read*, which is used to read attribute values from nodes. Each node in OPC-UA has a number of *Attributes*. The exact attributes for each node will depend on its NodeClass. These are typically things like *DisplayName*, *NodeClass*, *NodeId*, or *Value* for variables and *Abstract* (true if the type cannot be instantiated) for ObjectTypes. The exact attributes per NodeClass is found in section 5 of part 3 of the reference[35].

This service will also often access multiple source systems, since a single request can ask for a number of different attributes. The actual request itself is very simple, it consists of a list of ReadValueIds, as described in table 3.2.

Name	Type	Description
nodeId	NodeId	The NodeId of the node the attribute belongs to.
attributeId	Integer	The numerical id of the attribute, these are constant and defined in the OPC-UA standard.

Table 3.2: The ReadValueId structure in OPC-UA.

A few other options may be specified, in order to read part of an array structure, or return the time the attribute was last modified, but the core of the service is just this: a list of node/attribute pairs. Translating this to efficient requests can be complicated. Very frequently, a request

like this will contain many attributes from the same node. In that case it is very inefficient to request these one at a time from the source system. Instead, the server should group attributes belonging to the same node, and even group nodes belonging to the same source system, to make the requests to the underlying systems as efficient as possible.

Beyond that, there is no guarantee that all the different values for a single node exists in the same system. It is often the case that meta-data about some component is stored directly on the server, while the value of the sensor is stored elsewhere.

3.4 Timeseries and Event Data

Using the two services, Browse and Read, a user can access information about nodes. Most of the APOS information model would be accessible using just these two. There are, however, a number of other services that also warrant discussion, relating to reading data such as measurements and events from the source systems. The first of these is the *Subscriptions* system.

Subscriptions

Values in OPC-UA are often transient, as they reflect some state in an underlying system. A simple example is the value of some "Measurement" node, that reflects a live sensor value. Once this sensor value changes, the old value is by default lost, unless some other system makes an effort to store it. Even worse, events are often similarly transient. An event occurs, and is reported, but if no-one is listening, the event may in some cases be lost. The server could store values or events, but this is often not possible or desirable, as that would make it much less flat, and effectively add another source system for historical values.

The user could just periodically ask for updates to values, but this is not ideal. There is no guarantee that the connection between client and server is stable, or that the user is able to make requests quickly enough to sample the values they want. For example, if the signal the user wanted to measure had frequencies up to 1000 Hz, it would be difficult to reliably request updates over the internet. Even if the server is located close to the client, random network traffic, external disturbances, or load on the server would frequently cause requests to take too long. High frequency requests also adds unnecessary load on the network and server.

Instead, the user asks the server to sample values periodically, and store them until asked to *Publish* the stored values. For convenience, subscribed values are grouped together using *MonitoredItems*. Each *Subscription* has a number of MonitoredItems, which each monitor a single attribute on a single node. Each MonitoredItem either samples values at a specific rate, or receives updates in some other way (the details are left up to the server), and also queues a certain number of values. The client may request a specific *QueueLength*, for each MonitoredItem, which is how many values should be queued. The client calls the Publish service with the ID of the Subscription, and receives all stored values in response.

Using the 1000 Hz example, this is how it would typically work: The client first creates a Subscription on the server, and decides to call the *Publish* service once a second. It then creates a MonitoredItem attached to this Subscription, which samples at 2000 Hz, in order to capture all the high-frequency behavior. It sets the QueueLength of the MonitoredItem to 4000, to make sure that no data is lost, even if the client is not able to publish every second. Now, the server

will sample at 2000 Hz, and once a second the client will call the "Publish" service, and receive all 2000 measurements since the last publish from the server. This setup can be made to work even on unstable network connections.

Alternatively, the server might not sample at all, and instead update the subscription whenever the value changes, resulting in fewer data points. This is up to the server, when the client asks for a certain sample-rate, the server may not comply. Usually the server will indicate to the client a range of supported sample-rates.

Events

The fundamental system for reading events is similar, except it often does not make sense to "sample" events. Instead, they are reported to the MonitoredItem, and written to the queue when they occur. The MonitoredItem monitors the *EventNotifier* attribute on the node that generated the event, which is not necessarily its source. The structure of events is discussed in further detail in the specialization project report, chapter 3.

Events are in general much more complex than attributes, in that each event may contain a huge number of fields. Unlike subscriptions to attribute changes, subscribing to events requires the user to state which fields they want to read.

For example, the user might request a few core fields *EventType*, *SourceNode*, *Time*, and a property of a derived event type, *FailureMode*. It would then receive any events, even those that do not define FailureMode. In those cases, that field in the result would not be set. Each event received would consist of four values, in the same order as requested, regardless of whether the event type in question has the FailureMode property.

OPC-UA also defines a complex system for filtering, but a detailed understanding of that will not be covered here, and the server will not implement this beyond what is built into the SDK.

This general structure does make reading events more difficult. If the events were generated in a source system the server might need its own system for sampling events. The filtering system in OPC-UA is complex, and translating it to the filter system used by the source system, if any, may not be practical, so a server might have to just read all events, then just report the ones that are relevant to the client.

HistoryRead

The final group of services that may be worth considering for this project is reading historical data and events. This mostly works as one might expect. Each node indicate whether it has access to historical data (the *Historizing* attribute on variables), or historical events (The *HistoryEvents* flag on its *EventNotifier* attribute). The client may request historical information from a number of nodes at once, for specific periods of time, and using ContinuationPoints to continue reading, as discussed previously.

It is uncommon for low-level source systems to have access to historical data, but there is nothing preventing an OPC-UA server from using a more complex system like an external time-series database as a source system. In that case, reading live values and reading historical values might actually result in calls to *different* source systems.

3.5 Development of an OPC-UA server

Understanding the available services is essential for developing both client and server OPC-UA applications. This project aims to create a server that exposes multiple underlying systems, and a proper development procedure is needed. One such procedure is to go through each of the above services in the order presented here, and identify how to access that information in the interfaces of the source systems. This is often not simple, and the actual steps will vary based on the source system.

In order to illustrate how this might be done, this section goes through the process for a source system consisting of a database of employees in an organization. It is organized as a node tree, as that should be familiar to the reader, and most systems can be transformed into something similar. The structure is seen in figure 3.1.

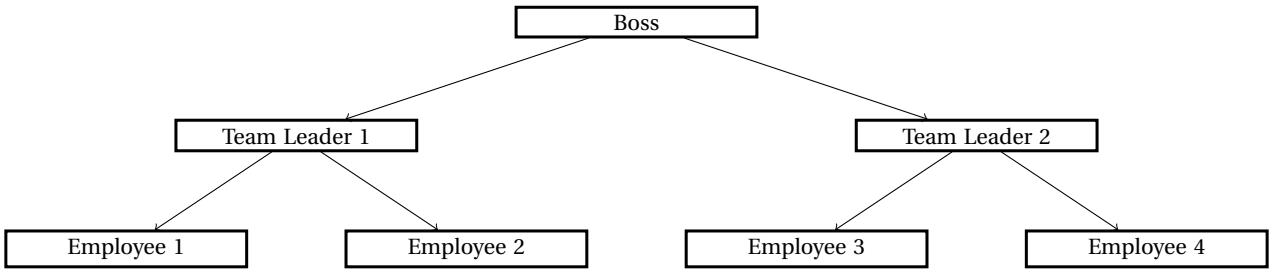


Figure 3.1: Sample structure from source system.

Browse

Starting with Browse. The first operation for a user might be to browse the “Objects” node, which is always at the root of the OPC-UA hierarchy. Browsing the *Objects* node would mean finding all nodes in the model without a parent, so the server queries the source system for any employees without a boss. This returns just the “Boss” node. Next the user asks to browse the “Boss” node, so the server responds by querying the source system for employees that work under “Boss”, resulting in “Team Leader 1” and “Team Leader 2”.

This is not, in practice, everything that would have to be returned by the Browse service. This database would be expected to also store information about each employee. For example, each employee has a “Firstname”, “Lastname” and “Address”. The OPC-UA attribute *DisplayName* can be used for Firstname and Lastname, but there is no obvious OPC-UA attribute that corresponds to Address. Trying to include it in the *Description* would not make it easy for an automated system to find the address, and would not make for a good information model.

Instead, in order to model this in OPC-UA one would use properties. Add a property to each employee node that stores the value of each employee’s Address. This is a separate node, so it should be returned when Browse is called.

So the Browse service call on the “Boss” node results in two operations: Query the database for employees that work under “Boss”, and retrieve information about the structure of the table containing “Boss” to know what properties to display. The second is going to be quite consistent, however, so it can be done on startup, to avoid unnecessary load on the source system.

The user might keep going, and Browse both “Team Leader 1” and “Team Leader 2”, where the server would give 6 nodes as a result: Employee 1 through 4, as well as “Address” for each of the two browsed nodes. Figure 3.2 shows the structure in OPC-UA.

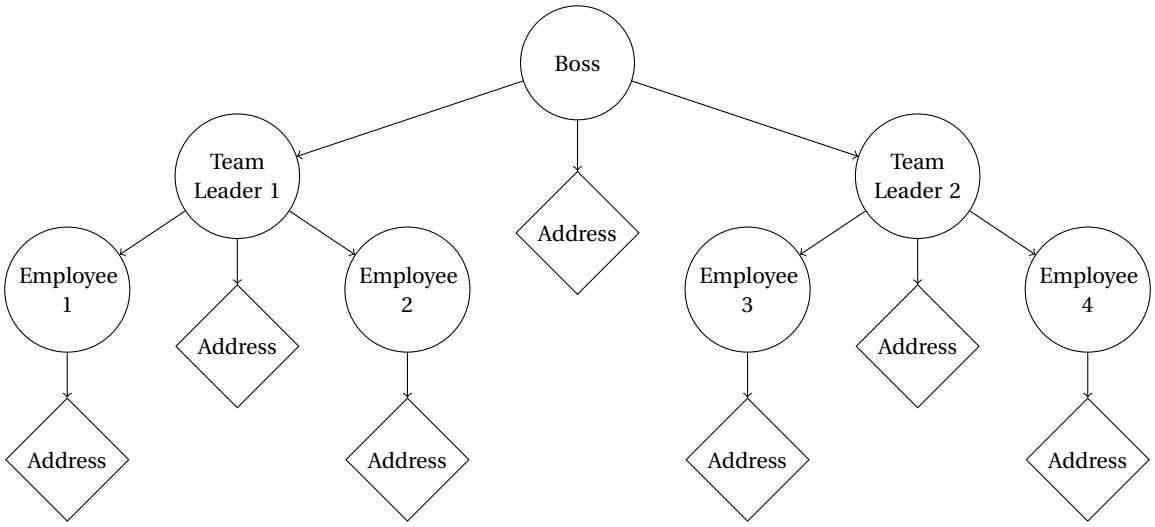


Figure 3.2: Structure in OPC-UA.

Read

Next the user invokes the Read service, to find out some more information about each node. Most of the returned information is defined by the information model, as described above. Employees do not have values, and represent instance information, so they have the “Object” NodeClass. The DisplayName of each employee could just be their first and last names. Many attributes are about permissions to edit, whether the node generates events, and so on. These can usually be known beforehand, and are not stored in the underlying system.

When a client asks about the “Address” node, the server needs to know how to access the value in the source system. Assuming the database uses numerical ids, each employee can be uniquely determined by some employee number. If, for example, “Boss” is employee number 1, the NodeId of the Boss node might be “ns=1;i=1”. The boss has an “address” field, which also needs a unique id. A solution is to let the NodeId of the address node on “Boss” be “ns=1;s=1-Address”, as this communicates a lot of information using the NodeId.

First, the namespace indicates that the source system is the employee database. Next, the identifier type indicates that this is a property of an employee. Finally, the identifier indicates that this is the “Address” field, and that it belongs to employee number 1. Since it is safe to assume that field names in the source system are unique in a given table, this means that all nodes discussed thus far have unique IDs.

Remaining Services

Next the user may want to access some other values. Perhaps management wants to add GPS trackers to the employees, and report that information in the database as a live value. The

client may subscribe to updates to this value, and so the server would need a system to poll the database for changes periodically.

Perhaps instead of simply making "GPS Position" a field on each "Employee" in the database, it could be written to a separate table, with a timestamp of the measurement, so that the user can access historical data as well. Now, although "Position" will still be a variable under each Employee node, the values are actually stored elsewhere, so when Read or HistoryRead is invoked on the *Value* attribute of that node, the server must look in a different table.

Finally, for convenience the server could trigger events each time an employee enters or leaves a dangerous area. This could be done using a similar system: when a client subscribes to events, subscribe to changes in position in the source system, then detect that the employee enters or leaves areas marked as dangerous, and report that as an event.

In practice, the actual implementation of the OPC-UA server should be fairly straight forward, given a good information model, and access to the source systems. If the general concepts and design philosophies described in this chapter are kept in mind, there should be few problems in the development of the server itself. A solid understanding of OPC-UA is important, however, as the standard is very flexible, and it is quite possible to create a poorly designed server. An additional advantage of a "flat" server is that expanding it to new source systems is usually fairly simple, so this should be a goal.

Chapter 4

Source Systems

As mentioned in chapter 1, the server will use OPC-UA to provide access to information in three separate underlying systems. These are all systems used by Aker BP engineers to organize information about the various facilities operated by Aker BP. In order to create a mapping from the source systems to the OPC-UA information model, it is necessary to understand how these source systems are used, what their role is, and what information they each contain that is of relevance to the APOS information model.

This project will focus on three different source systems in use at Aker BP. First, Aveva[8] LCI (Life Cycle Inventory), a database that contains a collection of tags, which each represent a single piece of physical equipment. The LCI database is the core of Aker BPs system, and connects to the other two systems. Secondly, EqHub[12], a central repository for equipment information managed by NOROG (Norsk Olje og Gass, eng: Norwegian Oil and Gas) and hosted by Sharecat[18]. Finally, SAP[17] (Systems Application and Products in Data Processing), a large CMMS (Computerized Maintenance Management System) used for, among a number of other things, managing maintenance and testing of equipment.

This chapter explains how these systems are constructed, how the data is accessed by engineers today, and how this project can access the data in question. Figure 4.1 shows the general structure of the server, and the type of interface available for each source system. This chapter explains in detail how each of the APIs below (REST, ODBC and ODATA) work, and how they can be used in the server application.

4.1 Process

From the beginning of this project it was not clear whether it would be possible to find any official APIs for any of the three source systems. In order to ensure the success of the project despite this, it was decided to reverse engineer the accessible frontend applications for all three systems, in order to find some way to access the information. The goal of reverse engineering the systems is to identify the internal flow of information in each application, and find out where it is "vulnerable", meaning where it can be diverted to a different application.

Reverse engineering any system, in order to gain access to the data it displays, typically follows the following general procedure.

- Find the desired information in a human-readable format. Since these applications are

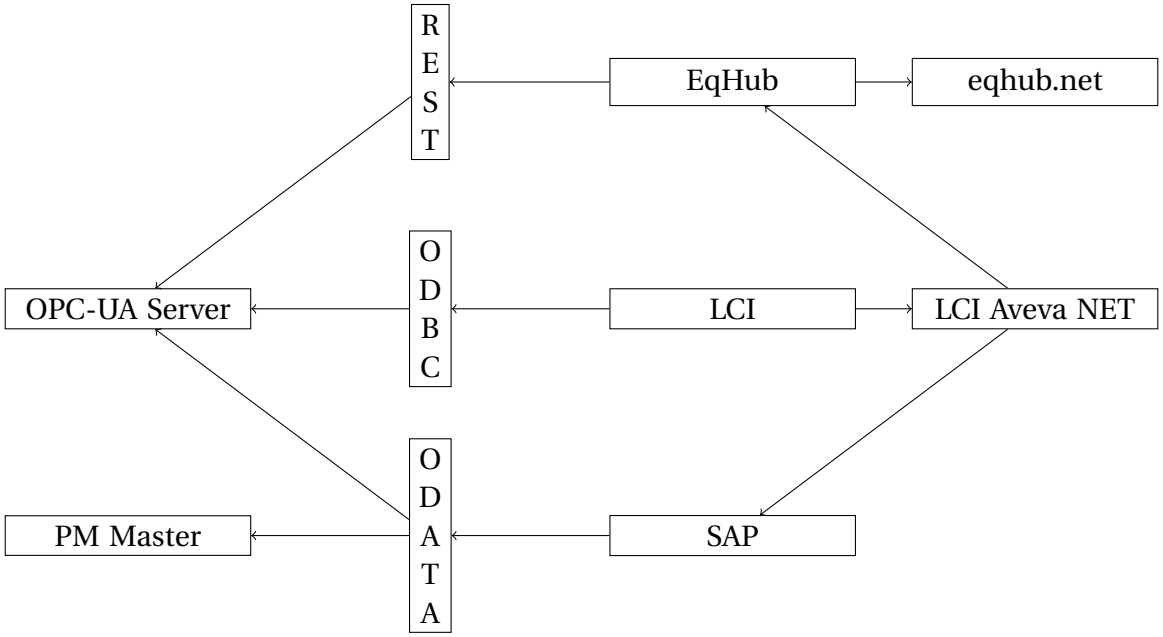


Figure 4.1: Source systems and their APIs.

designed to display information to users, this tends to be fairly easy.

- Identify where the information comes from, and where it can be intercepted while still in a machine-readable format.
- Study the request to find out where it goes, and what additional information is required to construct it.
- Repeat until a complete algorithm for obtaining the desired information from the server can be designed.

The idea is that most applications, for efficiency, communicate information on a machine-readable format, since this is often more compact and easy for the application to transform into something human readable. This is illustrated in figure 4.2.

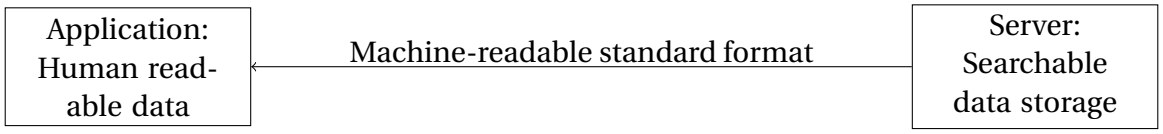


Figure 4.2: How most server/client applications are constructed.

All three source systems have some kind of web-based frontend application, which is advantageous here, as it makes reverse engineering much more likely to succeed. Web applications follow the general structure illustrated in figure 4.2, and run inside a container, the web browser. As such all requests from the web-application must be readable by the browser, so that the browser can decide how to send them over the internet. This uses open standards, and

messages are usually readable by both machines and humans. In order to understand how this works it is worthwhile to discuss some basics of the world wide web.

In general, communication on the world wide web[22] (WWW) is done over HTTP[27] (Hyper Text Transfer Protocol), which has a standard structure of a set of headers, and a body. HTTP itself uses TCP (Transmission Control Protocol) as a transport layer. Navigating to a webpage causes the browser to send a "GET" request to the server, where GET is a kind of classifier for the type of request, a part of the header. The server responds with whatever resource is located at the specified URL (Uniform Resource Locator). For most web-pages this will be an HTML[14] (HyperText Markup Language) page containing a number of references to other resources the server needs to fetch.

This data is used to build the page, trigger further requests, and in general display information to the user. Scripting on modern websites usually uses JavaScript[15], which is a programming language designed specifically for websites.

Requests are usually formatted as JSON (JavaScript Object Notation), which is convenient, as it is very easy to work with, and many programming languages have some kind of library to parse JSON. Requests can also be XML (Extensible Markup Language), which similarly can be easily parsed by both humans and computers. A website that uses some other, unusual format would be much harder to work with, but fortunately all three source systems use JSON, XML or HTML.

Given this, there are a few different parts of the site to study, in order to find out what information is available. The HTML that the page constructs and displays to the user can be studied to find out what information exists. The JavaScript can be studied to figure out what the website does, though this is often very difficult, since the JavaScript is usually "minified", meaning that it has been transformed to have a smaller size. This has the side effect of making it almost impossible to read. Finally, and most relevant for this project, the requests the website makes can be studied to find out what it communicates with the backend. Most modern browsers have tools for looking at the requests, as well as for studying the data stored in the web application.

Many websites, however, contain next to no JavaScript, and instead construct the entire page on the server, before sending it to the client. In this case, intercepting the data is much more difficult, since the HTML is designed to be human readable, not necessarily machine-readable. It is possible to do what is known as "Scraping", essentially just digging through the page to extract the data as it appears to the users. While HTML is designed to describe pretty websites it can still be read and parsed by a computer.

With the procedures outlined in this section, it should always be possible to extract data from a web application, in a more or less efficient manner. Even if there is no API, and requests cannot be intercepted, it is more or less impossible to design a website that can be read by humans but not by machines, so a computer program that impersonates a user would still be able to access the information.

Table 4.1 provides an overview of the results of this chapter, differentiating between the protocols and formats used by the frontend user applications, and the formats the OPC-UA server ended up using. These are each explained later in this chapter. For now, note that the user interfaces use exclusively human-readable formats: XML, JSON and HTML.

	User Interface		Available API	
System	Format	Protocol	Format	Protocol
SAP	XML/JSON	ODATA	XML/JSON	ODATA
LCI	JSON	REST	TDS	ODBC
EqHub	HTML	N/A	JSON	REST

Table 4.1: Overview of communication formats and protocols in the source systems

4.2 LCI

An LCI (Life Cycle Inventory) database is a system used to store information about equipment over their lifetime. It will typically contain information about instrument classes, safety requirements, location, logical relationships etc. In general, such a database is used to manage information related to equipment so that it is possible to look up related documents, measures and other instance-specific information.

User Interface

The first step to exploring this system is to look at how the data in the LCI database is accessed by users. In Aker BP this is done through an app called LCI AVEVA NET. The application itself does nothing out of the ordinary. When the user looks at a tag, it makes a few requests to the backend, that respond with JSON data. The requests are made with JSON payload to a .aspx endpoint. The .aspx file extension implies that this is an ASP.NET[7] application, which is expected given the name of the application.

This backend .NET application appears to combine a number of different systems, and present a common, if not particularly simple, API. The request contents are fairly simple, and could be reverse-engineered. The most unusual part of this system is that it seems like some requests, those that fetch SAP-related data, are made as JSON encoded SOAP, which is very unusual, and gives some important insights into the backend.

SOAP[3] (Service Oriented Architecture Protocol), is a Microsoft-developed standard for object access. It defines a message structure with an “envelope” around a “body”. It is older, but still widely used. Finding it in a system like this would not be surprising, however it is usually encoded as XML. In fact, it is defined within the XML standard. The response here is an XML structure encoded as JSON.

This further confirms that the backend here is an aggregator for several source systems. Likely this backend makes its own requests over a SOAP API to SAP, receives an XML encoded response, converts it internally to JSON, then sends that inside its own standard response body.

Finding something like this is not really a good sign for reverse-engineering this system. The fact that there are several hidden layers of API might mean that knowledge of internal logic is necessary to create something general. This logic may exist in the source code of the frontend, however this is minified JavaScript. That is, it is code that could in theory be read by a human, but it is essentially readable code that has been compressed by a computer to be almost completely unreadable. Finding even simple logic in something like that is a long and complicated project.

Instead, it is better to ignore the API backend for now, and look for the actual database itself. In the IT world, a database is often a quite well-defined concept, which it is likely possible to access remotely. This is indeed the case for the LCI system, which is a fairly standard SQL database.

Data Access

The actual LCI database in use at Aker BP is an instance of Microsoft's SQL Server, which means that accessing it in this case will mean using SQL over ODBC[33] (Open Database Connectivity). SQL ("sequel", Structured Query Language), is the standard language used for accessing relational databases. It was developed at IBM in the early 1970s[24], and is usually, depending on dialect, a fully capable programming language able to describe complex queries against a database. Unlike most programming languages it is *declarative*, rather than *imperative*, the difference being that where an imperative language describes the process required to reach some goal, a declarative language just describes the goal.

This text will not go into the details of SQL, the queries used will be explained when they come up. The general idea, however, is to describe the result as a filtered section of some table. A simple SQL query might be `SELECT Weight FROM Cars WHERE Manufacturer=Ford`, retrieve values from the "Weight" column for all rows in the "Cars" table where the value of the "Manufacturer" column is "Ford". The queries necessary for this project will not be much more complicated than this.

ODBC is a standard API for accessing databases. It is widely used, and Microsoft SQL Server databases will generally support it, the LCI database is fortunately no exception. Understanding the details of ODBC is not necessary for this project. The core is simply that it defines a standard SQL grammar, and rules for how messages should be structured. In this case, it gives access to the LCI database remotely. Communication over ODBC is done with the Microsoft format Tabular Data Stream (TDS), which is open and can be parsed with a library.

Unfortunately, the database structure for this particular database is not very descriptive. The database in question is called *LCI_Reporting_Prod*, and it contains a few *Views*, of which two may be of interest. A view is essentially a kind of virtual table. Since any SQL query returns a table, it is possible to perform queries against a stored query, a view, which can provide a convenient way to access information without having to look the underlying tables.

These views are called

- "REP_AssociationDetails", containing associations between tags, such as parent/child.
- "REP_TagAttributes", containing rows of "AttributeName", "AttributeValue", "TagName" and "TagId", making it possible to read the attributes of a given tag.

There is also a "REP_Tags" table, which is *not* a view, but which provides easier access to basic tag related data.

Some of what makes this database difficult is that the "AttributeName" for a tag is entirely numerical. So a simple query

```
SELECT [AttributeName], [AttributeValue]
    FROM [dbo].[REP_TagAttributes]
    WHERE [TagFullName]='Some|Tag'
```

would produce a table of pairs like "AttributeName=3033, AttributeValue=TS". The information for mapping these numerical names to actually useful values needs to be supplied from elsewhere.

A helpful engineer in Aker BP provided an SQL query with roughly 200 lines similar to:

```
,[22112] AS "22112 - EQUIPMENT IDENTIFIER"
```

which provides a mapping. A simple regular expression[6]

```
/\s*,\[([0-9]*)\][^-]*- ([^"]*)"/
```

is capable of parsing this to produce pairs of numerical attribute-name and a human readable name. The code for this is included in *LCIDBSource.cs*, *ParseRawNames*.

With these names it is possible to extract the data from the database as a list of name/value pairs, which can be assigned to the OPC-UA information model, as described in chapter 5. The associations table can be used to construct a hierarchical tag structure, and the tag-attributes table can be used to obtain properties for equipment instances.

A final point to note about the LCI database is that this particular database is just a copy taken at midnight every day, while the main database used by the LCI NET Application is inaccessible. Presumably, a full implementation would read directly from the main database, which is most likely identical to the copy.

Authorization

Gaining access to the SQL server is both easy and difficult in this case. It is easy in that any user in the same Active Directory (AD)[5] that has been given access, can read from the server. AD is a Microsoft Windows concept for user groups. This makes connecting to the server fairly simple, however it does mean that connecting to the server from a machine that is not managed by Aker BP is not really meant to be possible at all.

This would not be an issue, if it was not for the fact that the machine provided by Aker BP for this project is configured in a way that makes it close to impossible to compile .NET programs on it, meaning that development has to be done on a different machine and transferred there before running. Fortunately, this is not an insurmountable challenge, the setup used for this project is explained in some detail in appendix A.

4.3 EqHub

EqHub[12] is a service provided by Norsk Olje og Gass (Norwegian Oil and Gass)[1], which aims to gather equipment data in a central location. The idea behind this is solid from an information modeling perspective. Instead of each company like Aker BP having to get updated information about equipment from providers directly, it can be stored in a central repository and accessed using a common "TEK" number, which is found in the LCI database.

This is an initiative very much in the spirit of good information modeling. By gathering the information in a central repository accessed both by producers and consumers of industrial equipment, it is much more likely that everyone has access to up-to-date information about the equipment, which benefits everyone involved.

User Interface

The end user interface here appears to be another ASP.NET web-server. The actual equipment pages are linked from the LCI application, using numbers stored as tags in the LCI database, which are then linked to equipment tags using associations. Unfortunately, this time the site is mostly just a static page. That is, the actual information about each equipment type is written to an HTML document before it is sent from the backend server. This exposes almost nothing about how the data is stored. Most likely it is contained in some kind of database, but since the frontend does not use any kind of API, it is likely entirely internal, and since EqHub does not belong to Aker BP, accessing it was not possible for this project.

The upside is that the information appears to be organized in HTML tables, which means that it is reasonably systematic, and could fairly easily be scraped. The solution here does not have to be very complicated, but it is not ideal, and it would not be acceptable in a finished product.

Fortunately, there does exist a REST API which can be used instead.

Data Access

Accessing the information in practice means connecting to a REST API[11] managed by Sharecat[18], which is a Norwegian company that provides technical solutions for industry.

REST[28] (Representational State Transfer) is an architectural style of transferring data over HTTP. The idea is that the *state* is stored on a server, and specific operations modify or retrieve the data in specific ways. REST describes the behavior of four different HTTP *methods*:

GET Retrieve a representation of the state.

POST Let the target resource interpret the information sent in the request to execute some function on the state.

PUT Set the target state to the representation sent in the request.

DELETE Remove the state of the target resource.

Many modern websites loosely follow these conventions, and understanding them can make interpreting internet communication simpler, as they categorize the kind of effect each request has on the target state. For example, a GET request should not modify the target state, a POST request may, and a PUT or DELETE request should.

In this case, the REST API would make it possible to retrieve the data seen in the HTML tables directly, with relatively simple authentication. In general, since the TEK Number is already known from the LCI database, it should be sufficient to make a single request to the endpoint `/Catalogues/{catalogueId}/Teknos/{tekNoId}`, given the catalogueId, which is common for all TEK numbers.

The result is an easily parse-able JSON structure, which is documented in the open online reference for the API[11].

Authorization

Accessing the user interface can be done using Azure SSO (Single sign on), which is a solution for cross-platform authentication created by Microsoft. This, however, is not really designed for automation, and the REST API does not support it. Instead, in order to connect with the REST API, a special key has to be issued, in order to communicate over OAUTH 2.0[30].

OAUTH is a standard for authentication, in this case the system is fairly simple. A request is made using a client id and a secret key to an API endpoint, which returns a token that can be used to authorize requests for a limited period.

4.4 SAP

SAP[17] (Systems, Applications and Products in Data Processing) is a German company specializing in management software. The system in use at Aker BP is, in part, a so-called "Computerized Maintenance Management System" (CMMS), responsible for tracking testing and maintenance of instrumentation and other equipment.

One of the tasks of such a system is to keep track of function tests for equipment. The idea is simple enough: in order to ensure compliance with safety standards periodic tests are necessary. With tens of thousands of pieces of equipment of various types, a computerized system is needed to keep track of the enormous number of tests that will be performed over time. Performance is often measured in failure rate over time, so for example, if the failure rate is at most 1% per year, and 100 tests of an instrument type are conducted over a year, only one may be allowed to be in a failure state.

This results in an enormous number of data points across the system. A CMMS is responsible for organizing this data and contextualizing it with the correct tag numbers and other instrument information.

Of course, SAP does a great deal more than this. It is also used for requisitions, personnel, asset management, and much more. For this project, only three resource types are relevant: *Measurements*, as discussed above, *Notifications*, which are failure events with failure modes and detections, and *Functional Locations*, which correspond to LCI tags, and are used to connect tags in the LCI database with events stored in SAP.

User Interface

SAP is known in the industry to be quite complicated and it is essentially intended as an all-in-one solution for all asset management in an industrial company. In Aker BP access to SAP data goes through two primary systems. First is the SAP GUI, which is a standalone desktop application. This is unlikely to be possible to reverse engineer. It seems like it may be possible to access the data in SAP using the command line, but this is likely too large and complex to attempt in the limited amount of time given to this project.

The other is a web-app like the one for LCI, which is easier to work with. There are a number of different web apps accessible from Aker BPs "Fiori Launchpad". SAP Fiori[13] is a product developed by SAP which provides tools for developing a frontend application for accessing SAP. The actual name of the application is "com.akerbp.EamMasterApp", which points to this being

a product developed especially for Aker BP. The name used to refer to it in this text will be the PM Master App, which is the name used internally by engineers in Aker BP.

The app is somewhat unusual, and it does not appear to follow common guidelines for website design. Opening a single Functional Location produces over 250 requests, one of which appears to be a batch request that contains over 130 other requests. The majority of these are tiny JavaScript files that appear to define types, which has the consequence that the website appears to be slower than necessary, at least in terms of initial load times.

The actual data appears to be fetched in the aforementioned batch request, which makes requests to what seems to be an ODATA[16] API under the namespace "ZEAM_PM_MASTER_SRV". When the requests are made manually, the response is very slow, which indicates that the requests may be buffered. Alternatively, it might be that it is possible to make many parallel requests to the backend, even if each request is slow.

ODATA is a proper standard built on top of REST, it is formalized in ISO/IEC 20802. The idea is to describe a language for accessing, modifying, and querying data using REST. Essentially, ODATA defines a standard structure for requests making various operations, and legal responses to each such request. ODATA makes it possible to filter the response, query data for specific values, and transform the data in specific ways, by aggregating values.

Unfortunately, while it is possible to make requests directly to this ODATA API, it contains only a bare minimum of ODATA features, and it is going to be difficult to authenticate against. Using this would be a last resort, if no other API could be made available.

Data Access

As mentioned, ideally the data would be accessible through some official API. The best of these would be the official SAP ODATA API, which is well documented, and appears to be very powerful. The advantage for this use case is that if SAP had used some custom REST API, accessing it would require developing some custom application, like the one made for EqHub. For ODATA, however, there exists an official library for .NET which makes this a lot easier, if still not trivial.

Unfortunately, this system is not accessible at Aker BP. In fact, while some ODATA endpoints for limited subsets of SAP are in development and being gradually released in 2021, these will not come in time, nor will they cover the necessary data needed for this project. It might in theory be possible to read some data from official APIs, and some from reverse-engineering the app as mentioned above, but this will likely just take even more work, and be too complex for this project.

Instead, the best recourse is to try reverse engineering the PM Master ODATA API. This, unfortunately, is not going to be trivial, for several reasons.

1) There is no built-in system for authentication. Instead, Azure SSO will have to be used, which is not really intended to be possible.

2) There is no documentation.

3) It appears to be completely custom-built for this app, or at least for Fiori in general, and it is not a complete ODATA API.

Item 1) can be solved as described below, under "Authorization".

Item 2) is not a big issue. It makes developing a solution a bit more difficult, but ODATA contains a built-in system for API documentation, which is accessed by simply navigating to `odata/$metadata`, where the full list of types and endpoints will be listed. In addition, navigating to a page in the PM Master App means navigating to a URL with parameters matching the parameters to the ODATA API.

Item 3) is the biggest problem. If this was a fully fledged ODATA API it could be used to search, list and generally access data efficiently. However, this is not a fully fledged ODATA API, since it lacks collections. A core concept of ODATA is the idea of "Collections", which filters can be applied to in order to return a subset of the data in each.

For example, an ODATA request might look up all Cars in a database, this would look something like `odata/Cars`. This request should list out all Cars in a database, which is impractical. The theory, however, is that filters can be applied to this collection, so `odata/Cars?$search=ford` would apply a filter to the request to only retrieve a subset of the available data that matches the search term "ford". Filters can be very complicated, and return highly specific subsets of data. The alternative is to look up entries using their "Key Properties", which is a set of properties that uniquely define each entry, for example a car might be identified by its registration number, so a car could be found by navigating to `odata/Cars('REG-0001')`.

Without collections the only way to navigate is to know the key properties of each data type, and then using references. For example, a car might be associated with a list of repairs, and those could be seen by navigating to `odata/Cars('REG-0001')/repairs`. From there, each individual repair might be identified by a number, and the car it was associated with, so a repair could be found by navigating to `odata/Repairs('REG-0001', '1234')`.

While this is slightly more cumbersome than using collections, it would not be a very large challenge. The SAP Fiori ODATA API, however, uses a very large number of key attributes for each data type. Functional Locations have 9, and Notifications have 16. Some of these require a specific value, some can be left empty. This complexity means that despite using a standard API, it is still necessary to reverse engineer the application. In practice, the Fiori ODATA API does not consist of resource collections, but of functions which take many parameters, some optional, and return a list of results. This is a misuse of the ODATA standard, but it is not impossible to work with.

Authorization

It is possible to access the ODATA API directly when logged in to SAP through Azure SSO. Authentication on the internet, in most cases, will involve one or more values in the HTTP Header of each request. In this case, looking at the requests made to the server reveals that the only candidates are some SAP specific cookies, which are likely obtained through Azure authentication. Cookies are small, named, data packets that help persist information across requests on the internet. The client stores cookies as requested by the server, and the server uses these for authenticating and identifying the user.

In order to reverse engineer this, the first step is to make a request to the server without any cookies. It first redirects to a local page in the PM Master App, which then uses JavaScript to redirect to a Microsoft online login page. Here, the user *may* be prompted for username and password, unless they are accessing the site from a computer on the same Active Domain as the server, such as the PC provided by Aker BP. After credentials have been entered somehow, the

site will produce a Single-Sign On (SSO) request, then use JavaScript to poll for a response for about a minute, after which it times out.

Once this is done, the client is redirected back to the PM Master App, and are given SAP cookies that can be used for authenticating requests to the API for a few hours. In order to automate this process, it is necessary to run what is known as a "Web Driver", a kind of stripped down browser, which the OPC-UA server can communicate with. This is effectively a way to impersonate a user. It requires manually accepting the SSO request when first starting the server, but after the initial brief interaction, the results can be reused automatically for a few hours. By saving the credentials to a file, this process can be performed only about once a day. The process is illustrated in figure 4.3.

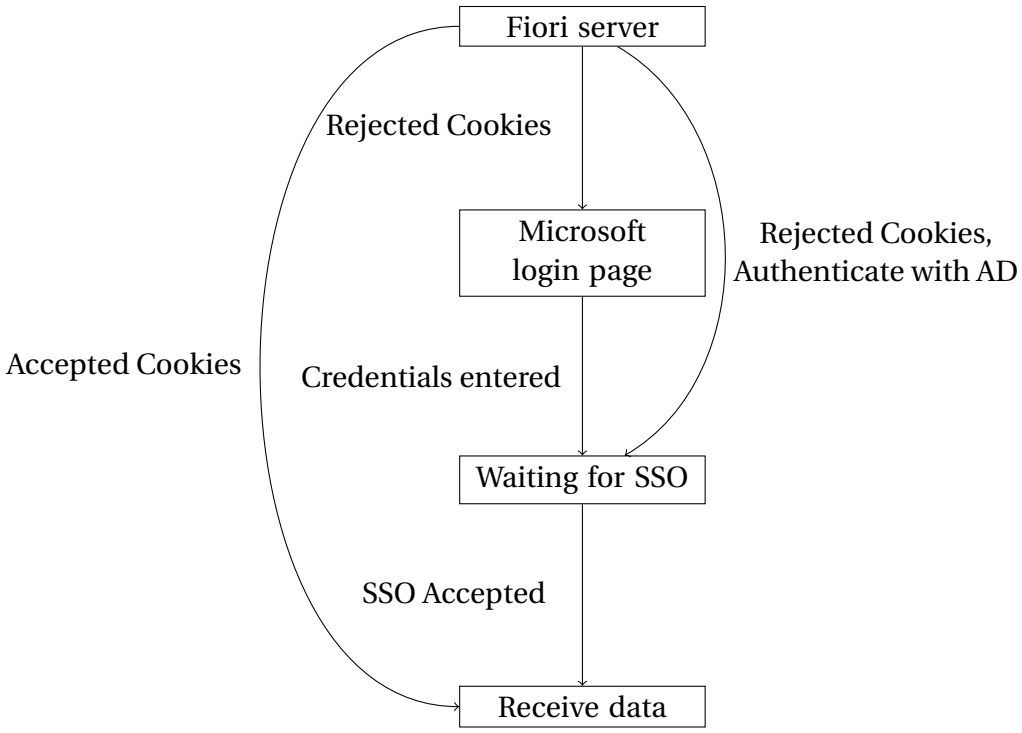


Figure 4.3: Authentication flow in the Fiori App.

This solution, despite being complicated and unreliable, is sufficient to create a working proof of concept for this project. A proper implementation would most likely have to find some other way of accessing the information, using a proper system for authentication.

Essentially, this solution is like wanting to trigger an alarm which requires an electric signal, without being able to access the wire at all. As a workaround, you create a robot that enters the building and pushes a button on the operator panel. Much more difficult and unreliable, but it can be made to work. Most of what humans do on a computer can be recreated by a machine, but doing it the "human" way is usually the last resort.

Chapter 5

Information Mapping

In order to consistently and repeatably convert the data from the source systems into the APOS information model, it is necessary to produce a machine- and human readable mapping. For the purposes of this project, this is done by creating a simple configuration schema describing mapping from source systems to the APOS model.

Not all mapping will require a configuration schema. All this logic could in principle be written directly into the code, and for some data-types this makes sense. The advantage of a configuration schema is that it is easier to read, easier to edit, and that it can be modified without re-compiling the code.

The configuration format should be simple and readable, and so the language YAML[23] (YAML Ain't Markup Language) will be used for all the custom configuration for this project. When creating a configuration schema the usual design philosophy is that it is more important that it can be easily understood by humans than by machines, and the schema created here will follow that principle. Beyond that, to reduce the amount of manual work, it will be made as compact as possible without sacrificing readability.

5.1 EqHub

EqHub only provides one kind of information: general equipment data. This is explicit in the information model, a single equipment type in EqHub is usually linked to many tags in the LCI database. This maps well to APOS equipment types. The L3 types as described in the specialization project are intended to refer to specific equipment models, so the only mapping necessary here is deciding what category each equipment model should belong to, and which attributes should be transferred from EqHub.

The processed data from EqHub is a long list of key/value pairs for each TEK number, with fields like "Class", "Supply Voltage Range", etc. "Class" is the closest to an APOS classification, and some classes match APOS L2 groups, but for other classes the EqHub types are too unspecific, so a configuration schema is needed.

Grouping

In order to solve the issue of grouping equipment types, a reasonable approach is to study how a human would approach classification, and attempt to create a configuration schema that re-

Name	Value
Model	PIR7000
Class	Detector, Gas
Type of gas	Hydrocarbon
Gas detector type	Infrared
Number of gases	Single gas
Description	The Dräger PIR 7000 is an explosion proof point infrared gas detector for continuous monitoring of flammable gases and vapours. With its stainless steel SS 316L enclosure and driftfree optics this detector is built for the harshest industrial environments, e.g. offshore installations.

Table 5.1: Some fields in an EqHub gas detector.

flects that process.

For example, if a human wanted to classify a gas detector using the properties in table 5.1, they could start with looking at the class, which immediately places it into the L1 group GasDetectors. Next, they might use the description to classify it further, but the description is generally not structured enough to analyze consistently using a computer program. Instead, the fields "Type of gas" and "Gas detector type" contain enough information to place it into the "PointHCGasDetectorIR" group.

For other types the process is considerably simpler. EqHub has a "Transmitter, Pressure" class, which maps directly to an L2 APOS group.

This approach can be seen as a form of *Rule based modeling*, meaning that it is just a list of rules that are applied sequentially until one matches. The configuration will start with the strictest rules, then apply less and less strict rules sequentially. The syntax used is a YAML list, elements are indicated by dashes, where each entry contains the name of the group it is specifying, and a number of fields with a list of requirements each.

```
- name: PressureTransmitter
  fields:
      Class: ["Pressure", "Transmitter"]
- name: PointHCGasDetectorIR
  fields:
      Class: ["Detector, Gas"]
      Type of gas: ["Hydrocarbon"]
      Gas detector type: ["Infrared"]
- name: GasDetectors
  fields:
      Class: ["Detector, Gas"]
```

This list can be translated into the following manual application of rules:

```
If the ‘‘Class’’ of the equipment contains ‘‘Pressure’’ and ‘‘Transmitter’’:
    It belongs to the PressureTransmitter group.
Else, if the ‘‘Class’’ contains ‘‘Detector, Gas’’
    and ‘‘Type of gas’’ contains ‘‘Hydrocarbon’’,
    and ‘‘Gas detector type’’ contains ‘‘Infrared’’:
```

```
        It belongs to the PointHCGasDetectorIR group.
    Else, if the ‘‘Class’’ contains ‘‘Detector, Gas’’:
        It belongs to the GasDetectors group.
```

Equipment that fit none of these rules are rejected and reported, so that the configuration schema can be manually extended to include them. Note that this way, all gas detectors are classified to the GasDetectors L1 group, ensuring that none are lost.

Attributes

The simplest solution to mapping the attributes over is just taking all of them and adding them directly. However, the model defines a list of attribute groups like “MeasuringPrinciple” or “DesignMountingPrinciple”, and the configuration schema could support this.

There is no good way to do this automatically, so the easiest is to just assign each attribute group a list of fields. This assumes that if attributes exist on multiple equipment classes they still belong to the same group, which seems likely.

MeasuringPrinciple:
- **Gas detector type**
- **Number of gases**
- **Concentration range**

DesignMountingPrinciple:
- **Supply Voltage Range**

The question really is if these categories fit very well at all. The APOS model is focused only on properties that are relevant for the automated safety follow-up. When designing a system like this in practice, there is very little reason not to also include other relevant information. A good solution might be to use standard names or groups for the few attributes that are most relevant to APOS, and just add all the others outside of any category, so that no information is completely lost.

Identifiers

In OPC-UA, an essential decision is how to make unique identifiers. Doing this intelligently is important, since it is quite possible with OPC-UA to request attributes from just a single property of an equipment type. I.e. a user might request the “Supply voltage range” of a specific equipment type. This would be a single property-typed node in OPC-UA, and the request would contain only the NodeId of the property itself, without the NodeId of the type.

The equipment type itself can just use the TEK number combined with the type in EqHub, which is either “TEK” or “SPC”. A single namespace for EqHub should be sufficient, given the index “3” here to illustrate. The NodeId of the equipment type with TEK 00670799 would then be “ns=3;s=TEK-670799”. This ID supplied to the server would be sufficient to look up the TEK number in the EqHub API.

Next, the ID of each property should encode both the TEK number of the equipment type it belongs to, and the name of the field. For example, “ns=3;s=TEK-670799-SupplyVoltageRange”. This value contains the necessary information to retrieve the equipment type from EqHub, and

identify which attribute is requested. If the attribute does not exist, or is excluded, the server can respond with an error.

5.2 LCI

The LCI database is in many ways the link between the different systems, and will form the foundation of the instance hierarchy. The database contains the full list of tag instances with their attributes, and an association table that connects different tags in child/parent ids.

The processed data for a tag comes in two forms: A list of key/value pairs that correspond to the attributes and tag information, and a list of relationships with other tags/groups. This will be used to construct instance specific information and its references to other nodes respectively.

Attributes

The LCI database define over 180 attributes for each tag, of which most are unused and empty. In the LCI app, a lot of these are hidden or ignored. For this project an acceptable solution is to simply ignore empty attributes. Next, it makes sense to exclude specific properties that duplicate information stored in EqHub, or only make sense in the context of the LCI application. The schema for this can be just a single list of attribute names.

```
ExcludedProperties:
  - "CrossSectionalArea"
  - "InsulationClass"
```

References

In general, there are few references for each node. Most seem to just have two: a parent/child reference to another node, which translates to a hierarchical reference in OPC-UA, and a grouping reference to a TEK number, which translates to a reference to an equipment type.

There are few enough types here that creating a configuration schema is unnecessarily complicated. This should just be hard-coded. If there were other, more complicated relationships between tags in the database a mapping between LCI reference types and OPC-UA reference types might be needed, but that is not the case.

TEK Numbers are also, in fact, tags in the LCI database. This means that the LCI database can be used to read reverse references from EqHub nodes to equipment, which is valuable, as it makes it possible to find information related to all equipment of a specific type.

Identifiers

NodeIds can be created using the same general strategy as for EqHub equipment. Tags in LCI are defined by their *FullName*, which is on the form “TAG|LOCATION|NAME”. This is somewhat less general than a number, so in order to name properties intelligently, it is necessary to use a different separator. Using“-” again could cause issues when adding attributes to tags with dashes in their names. For example, creating the NodeId of a property named “some-value” on the tag “TAG|ULA|GD-0001”.

Using the same strategy as for EqHub, connecting them with a dash, like “TAG|ULA|GD-0001-some-value”, there would be no good way of knowing whether this was a property named “value” belonging to “TAG|ULA|GD-0001-some”, or some other permutation. Instead, using “|” as a separator should create consistent results. This produces the ID “Tag|ULA|GD-0001|some-value”, which should be general enough. This does assume that tag names are not allowed to contain “|”, which is not known for certain, but appears to be the case for all tags observed in this project.

5.3 SAP

While there is some tag data in SAP, most of this exists in some form or another in LCI or EqHub. For the purposes of this project, SAP will only be used to produce Events. There are two types of event-like data which will be represented as OPC-UA events. Measurements and Notifications.

Measurements

Measurements come in the form of a list of short measurements linked by the FunctionalLocation, and a larger *MeasurementDocument* for each short measurement. A measurement is not a live data reading, those are not stored in SAP, but instead a periodic, manual functional test to ensure that the equipment is still working. This is not really mapped to APOS in any way, but it does tie into Notifications, and failure rate analysis is in general relevant for APOS.

The way this is done is simply by picking the fields that seem valuable, then mapping those to event properties in OPC-UA. Since events in OPC-UA must declare all the fields they are providing, the event must be created beforehand, and all relevant fields must be added there. Here it is also possible to make a few inferred fields. In particular something like a boolean “Pass”, which would be useful for filtering purposes.

Beyond that, when designing an event in OPC-UA, some fields may be mapped to the existing fields on the BaseEventType: “EventId”, “EventType”, “SourceNode”, “SourceName”, “Time”, “ReceiveTime”, “LocalTime”, “Message”, “Severity”.

- *EventId* in this case needs to combine two pieces of information: The Measuring Document number, and the Measuring Point number. Fortunately these are both numeric and hence on a simple form. EventIds are byte-strings, so a solution is to treat the two numbers as 64 bit, 8 byte numbers, then chain them after each other to form an 128 bit, 16 byte string.
- *SourceNode* and *SourceName* describe the node this event originated from, which translates to the tag connected to the Functional Location this measurement belongs to.
- *Time* is equal to *LocalTime* and *ReceiveTime*, and is given by the fields “MeasurementDate” and “MeasurementTime”, which can be combined into a single date-time value.
- *EventType* is given by the NodeId of a “SAP Measurement” event type, which is a sub type of the BaseEventType node.

- *Message* can be constructed from the measurement description, type and result. These can be added separately as properties on the custom event, but keeping them here makes it easier to see at a glance what the event is about.
- *Severity* is not specified here, but for simplicity it can be set to "Low" for passes and "Medium" for failures.

Finally, a decision must be made as to which node generates each event. Since the events are read from the source system (SAP) through the tag itself, it is reasonable that the tag also generates the event. There is no good way to "subscribe" to events here, and they are very infrequent (about once a year), so each node will only support reading historical events.

Notifications

Notifications are the closest equivalent to failure events in SAP. They have failure modes and detections, which can be mapped to APOS. Notifications are generated either manually by operators, for example if a measurement gives a negative result, or automatically by the device itself, if it reports some issue.

Reading them is done in the same way as measurements. Each Functional Location is linked to a list of basic notifications, that give the ID of a full notification.

A good solution for this is to create a sub-type of the APOSFailureEventType in the information model. The mapping of the base properties is more or less the same as for measurements, except for Severity, EventType and EventId. Notifications provide a "Priority", which can be mapped to Severity, and the EventType is obviously now the new type, which is a sub-type of APOSFailureEventType. Notifications are given by just a single number, so it seems reasonable to let the EventId just be the notification number converted to 8 bytes.

Actually mapping the failure modes and detections is slightly harder. Modes are given as three-letter codes, and detections are given as numeric codes. SAP of course uses a different system than APOS, so the mapping is defined in a configuration schema.

FailureModes:
SER: MinorInServiceProblems

FailureDetections:
MADM0001−0005: Diagnosed/ImmediatelyDetectedEvent

It seems like some failure detection descriptions match the ISO-14224 failure detections, which are mapped to APOS in the H1 report. MADM0001-0005, for example, is described as "Continuous condition monitoring", which is detection method 06 in 14224, and corresponds to "Diagnosed/Immediately Detected Event" in APOS.

It unfortunately seems very common to have failure modes/detections that are simply OTH, "Other", which is so general that there is no way to really do a proper mapping. This project will just map those to "MinorInServiceProblems" for failure modes, and "CasualObservation" for detections, even though that implies some periodic check. It could also easily be "Other PM Activity". "Other" could be treated as undefined, and not mapped at all, but this too would not be entirely accurate. For example, if a notification has detection method "Other" it almost

certainly means that the failure was not detected automatically, which excludes a number of detection methods.

Chapter 6

Implementation

The server implementation itself is written in C#, using the official OPC Foundation SDK[20] for the OPC-UA part of the system. In addition, the open source library YamlDotNet[2] is used to handle YAML configuration. The other libraries used (System.Text.Json, Linq, etc.) are part of the core .NET libraries.

This chapter will go through the implementation and illustrate the choices made, without going into great detail on the actual code. Although the code is provided along with this project, it is not going to be possible for most people to actually replicate the results, even with detailed instructions included in appendix A, since it requires access to internal Aker BP systems. Hence, the various functionality will be shown in figures taken from UAExpert in appendix B.

6.1 Processing data from Source Systems

The program is divided in two functional systems, where the first is the code for fetching and parsing raw data from the source systems. The general concepts are described superficially in chapter 4, but this section will go into the actual decisions made and discuss each type of request the server makes to the source systems.

LCI

LCI is, as mentioned in previous chapters, the core of Aker BPs system. It references EqHub nodes through TEK Numbers, and it also provides the necessary information to identify SAP tags. Requests here are made with SQL over ODBC, as discussed in section 4.2.

There are three primary forms of data obtained through the LCI database: Tag data is basic information about each tag, like its name, ID and class in the LCI database. Tag attributes refer to complex data related to each tag, like location, facility and function. Finally, tag relations refer to relationships to other tags in the LCI database. Since TEK numbers are also tags, this includes the equipment type of each tag.

Tag data

Fetching the tag first helps make more efficient requests later. The NodeId is built from the "FullName" of the tag, on the form "TAG|SITE|Name", but it is more efficient to use the internal

id of each tag when fetching attributes and relations later, this ID is obtained by fetching from the table "REP_Tags", using the FullName. The query is

```
USE [LCI_Reporting_Prod]

SELECT [TagDescription], [TagID]
FROM [dbo].[REP_Tags]
WHERE [TagFullName]='{tagName}';
```

This query is very simple, it fetches three values "TagDescription", "TagClassName" and "TagID" from the table named "REP_Tags". TagDescription is used as the description of the OPC-UA node, TagClassName is useful to know what the node describes, like whether it is a TEK number or a physical device, and TagID is the internal ID used for queries later.

Tag Attributes

Tag attributes are fetched using the TagID. This ends up being by far the heaviest query made to the LCI database. The attributes are just TagID/AttributeName/AttributeValue, where AttributeName is numeric. The final result uses a configuration file created as described in section 4.2 to map AttributeName to a human readable name.

For efficiency, the solution ended up not using the view "REP_TagAttributes" mentioned on page 27. While views are convenient, the database in question is quite slow, and since the view definitions could not be obtained, they may carry some unpredictable performance costs, even if they really should not. Instead, the query used replicates the functionality of the view, but removes a few fields that were not needed for this project.

```
USE [LCI_Reporting_Prod]

SELECT [AttributeName], [AttributeValue], [AttributeUom]
FROM [dbo].[REP_AttributesOfTags]
WHERE [TagID]={tagId} AND [AttributeValue] IS NOT NULL
```

This query simply fetches all attributes which have the given TagID, and which are not empty. Most attributes are empty, and sending those over is, in this case, a waste of resources. The fact that so many fields are empty is also probably part of the reason why this particular table is so slow. Most of its rows are empty, so the database has to look through a very large number of rows to find the actual information.

The "AttributeUom" field has not been mentioned before. Some fields have a "Unit Of Measurement". If they do, it can simply be combined with the value in the result displayed to the user.

Tag Relations

The full relationships table in the database contain an enormous number of relationships, connecting tags to documents, documents to documents, tags to tags, and likely more. For this project only tag to tag relations are needed, since documents are not mapped to OPC-UA.

Again, while the view "REP_AssociationDetails" is convenient for making a few queries, it is very slow, and so it had to be reverse-engineered to use the TagID instead of the TagFullName.

This manual query is about 10 times faster on average, and is most likely functionally identical to the view, since it returns the same total number of rows.

```
USE [LCI_Reporting_Prod]

SELECT [SourceID], [TargetID], [AssociationId],
    source.[TagFullName] as [SourceName],
    target.[TagFullName] as [TargetName]
FROM ([dbo].[REP_AssociationIds]
INNER JOIN [dbo].[REP_Tags] as source
    ON source.[TagID] = [SourceID]
INNER JOIN [dbo].[REP_Tags] as target
    ON target.[TagID] = [TargetID])
WHERE [SourceID]={tagId} OR [TargetID] = {tagId}
```

This query is a fair bit more complicated, but it should be possible to understand. It uses the SQL "INNER JOIN" operation, which is shorthand for the combination of two operations: cross product and filter. Essentially, first produce all possible combinations of rows in both tables, then filter them using the "ON" part of the operation. In practice, the database is smart enough to be far more efficient than this, but that is the basic theory.

This does two INNER JOINs to the same table. So essentially: find all associations in the "REP_AssociationIds" table where the Source or Target matches the provided TagID. Next, match both sides of the association to a row in the "REP_Tags" table, then remove all that do not have a match.

The final result is a relation that includes the full name and IDs of both the target and the source, as well as the "Id" of the association, which is a three-letter code that indicates what the relation represents. There are four relation types in this table that may be relevant to this project:

ASS "is part of" and **ICO**, "is logically connected to", which connects components. These are non-hierarchical, but they do not seem to appear for any of the components looked at in this project yet. They can be mapped to non-hierarchical reference types in OPC-UA.

PAR "is standard part for", which is the association between tags and TEK numbers. These become the ISA95 reference HasPhysicalAssetClass and connect EqHub nodes and LCI tags.

PNT "is parent of", which is the normal parent/child hierarchical relationship. These are mapped to "Organizes" in OPC-UA, which is a hierarchical reference type often used for this kind of relation.

SAP Tag

Finally, it is also useful here to create a query to fetch the SAP tag name without reading all attributes of an LCI node, to make that process a bit faster. The query for this is just the full attribute query described on page 42, with `AND [AttributeName]=35390` added to the end. 35390 is the AttributeName of the Platform Code attribute, which is used to construct the SAP tag name.

Internal Structure

Each LCI node has a list of relationships, a list of attributes, and some core metadata. This should be represented internally in the server in some way, and since C# is an object oriented programming language the idiomatic way to do this is by creating an object graph seen in figure 6.1.

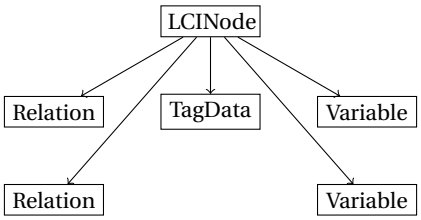


Figure 6.1: LCINode object structure. Each box is an object.

This may seem trivial for now, but as the internal state is expanded it is increasingly valuable to organize the data in a logical and intuitive manner. The code for populating the contents described here is included in "SourceSystems/LCIDBSource.cs".

EqHub

Reading data from EqHub means serializing requests to the REST API, then deserializing the result. There are two main API calls that are relevant for the core application:

First, fetching the authentication token is done using OAUTH, which is fairly simple, and covered well by the official documentation[11]. It essentially just means making a single request containing the secret key and client id to the OAUTH endpoint.

Next, fetching the data for a given TEK number is done with a call to the endpoint `Catalogues/[catId]/Teknos/[tekNo]`, where "catId" is the catalogue id, which can be obtained from another endpoint, but is constant that can be stored in code. The result is a JSON structure described in the documentation, which is deserialized using the System.Text.Json library.

This is done in two steps. The result is first deserialized to an object which mirrors the response payload, often called a "Data Transfer Object" (DTO). This is converted to an **Equipment** object which organizes the data on a format suitable for the server application. These can be used to create OPC-UA nodes later.

SAP

SAP is considerably more complicated. As mentioned, the ODATA API is reverse engineered by looking at the PM Master app as described in section 4.4. This section describes the three different types of data retrieved from SAP: *FunctionalLocations*, *Measurements*, and *Notifications*. The functional location itself is not used, but it is needed in order to obtain the measurements and notifications linked to by each tag.

From the LCI database the SAP tag name on the form "SITE-Name" is obtained. SITE refers to the *Platform Code* attribute in the LCI database, and represent different parts of facilities oper-

ated by Aker BP. For example, the ULA platform has sites ULP, ULQ and ULD. So “TAG|ULA|GD-0001” maps to SAP tag “ULQ-GD-0001”.

Functional Location

The functional location is retrieved using the FunctionalLocationSet ODATA endpoint, which has 9 key attributes:

- *Parent* and *FunctionalLocation* are both set to the SAP tag name obtained from the LCI database.
- *ExpandToLevel* is set to 0, and *FLMultiple* is left empty. It is not clear at all what these do, and it is not really important.
- *PlantFilter* needs to be set. The value is given by the number given to the planning plant. For ULA MPP this is 5000. A section of the configuration file maps the three-letter platform codes to the numerical planning plant value, for convenience. All three ULA platform codes, ULP, ULQ and ULD, use 5000, and the codes for other platforms can be found in the PM Master app if needed. The number also needs to be followed by a %2C, which is an encoded comma, for some unknown reason.
- *MainPlantFilter*, *StatusExFilter*, *StatusFilter*, *WithStatus* and *WithChar* can all be left empty.

Filling in the fields as described above produces the following query:

```
FunctionalLocationSet(
    Parent='ULQ-TagName',
    FunctionalLocation='ULQ-TagName',
    ExpandToLevel=0,
    FLMultiple='',
    PlantFilter='5000%2C',
    MainPlantFilter='',
    StatusExFilter='',
    StatusFilter='',
    WithStatus='',
    WithChar='')
```

Like the EqHub API, the response is deserialized to a DTO, which is then converted into a class defined in “FunctionalLocation.cs”. This result is not used in the final server, but the code to deserialize it was kept, in case it is needed in the future.

Measurements

Using the functional location URL the measurements can be retrieved by following a link, resulting in the query

```
FunctionalLocationSet(...)/to_meas_doc
```

This produces a list of very basic measurements, which each include a *MeasurementDocument* and *MeasuringPoint* number, which can be used to query the *MeasurementDocSet* ODATA endpoint, which only uses three key attributes.

```
MeasurementDocSet(
    MeasDocument='...',
    MeasuringPoint='...',
    FunctionalLocation='ULQ-TagName')
```

Again, the result is deserialized to a DTO, and converted into a proper **Measurement** object. This process converts the timestamp, which is in a SAP specific format, to a general C# DateTime structure.

Notifications

Notifications are similar to measurements, in that the first query is simple:

```
FunctionalLocationSet(...)/to_notif_main
```

The result contains a *Notification* field, which is used in the next query. The *NotificationSet* endpoint takes 16 different key attributes:

- *Parent* and *FunctionalLocation* are the same as before, the full tag name.
- *Notification* and *NotificationNumber* are equal, and should contain the Notification value mentioned above.
- *ExpandToLevel* is set to 0 again.
- *PlantFilter* is the same as for FunctionalLocation.
- *FlMultiple*, *StatusExFilter*, *StatusFilter*, *WithChar* and *UserStatusExFilter* are all left empty.
- *Phase1* is set to 9, for some unknown reason, it is not clear what this means, but 9 works fine. *Phase2* is left empty.
- *WithStatus*, *WithRepair* and *WithDetection* are set to 'X'. Presumably the "With" fields are used to filter out parts of the notification. Since knowing the detection method is valuable, these are kept.

This is the more or less the same configuration used by the PM Master app. It is possible that more information could be obtained by experimenting further, but the result contains sufficient information to construct the notification events later. The full query is

```
NotificationSet(
    Parent='ULQ-TagName',
    Notification='...',
    NotificationNumber='...',
    FunctionalLocation='ULQ-TagName',
    ExpandToLevel=0,
```

```
FLMultiple='',
PlantFilter='5000%2C',
Phase1='9',
Phase2='',
StatusExFilter='',
StatusFilter='',
WithStatus='X',
WithChar='',
UserStatusExFilter='',
WithRepair='X',
WithDetection='X')
```

which produces readable results. The result is mapped to a data transfer object, and combined with the basic notification linked by the functional location to create **Notification** objects containing the necessary data to create events later on.

Internal Structure

In the end, it is convenient to keep these together in a **SapTagData** class, shown in figure 6.2, which makes it possible to cache the results. Ideally it would not be necessary to cache anything, but fetching all this data for a node with a notification or two takes as much as 10 seconds, so for the purposes of this project, caching may help speed up successive queries, and make it more possible to find uses for this data later.

The advantage over the queries to the LCI database is that they can be made in parallel. Making 10 requests in parallel results in a time of about 20 seconds per node, which is still 5 times faster than querying them one at a time. Larger numbers of parallel queries have diminishing returns.

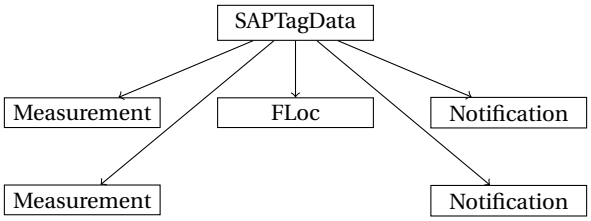


Figure 6.2: SapTagData object structure. Each box is an object.

6.2 Server

The second part of the program is the OPC-UA server itself, which calls methods on the SourceSystem managers with simple arguments, and obtains the transformed data classes described above. In the SDK, custom behaviour is primarily added by adding *NodeManagers*. This project implements two different node managers. One for internal nodes, and one for external nodes.

Internal nodes

The **InternalNodeManager** extends the **CustomNodeManager2** type from the OPC-UA SDK. This is used for creating very simple in-memory node managers, with little custom behavior. It is not sufficient for the external nodes, but it is suitable for creating a static basis for the type hierarchy.

The first step is to replicate the work done in the specialization project, the APOS information model. This could in principle use the NodeSet2 file created there, but doing this creates some issues. The NodeIds used there were numeric, and it is impractical to refer to numeric ids for everything. Referring to the APOS types should be very simple, so they should have NodeIds equal to their name. Rather than manually modifying the NodeSet2 file, it is edited to remove everything except the core types, then describe the equipment and failure classifier hierarchies using a configuration file.

The file is included, "aposstructure.yml". The equipment hierarchy is a map of objects, so that each type has the name of their parent and a list of allowed attribute groups, and the failure classifier hierarchy is a map from child to parent.

This recreates the model, without having to manually create the NodeSet2 file. It makes the model less shareable, but the code could be modified to generate a NodeSet2 file containing the generated hierarchy.

In addition, the two new event types are added, **SAPMeasurement** and **SAPNotification**, as well as two reference types corresponding to the two non-hierarchical tag relation types mentioned above, **IsLogicallyConnectedTo** and **IsPartOf**.

The event types have their own set of properties, but these are just taken directly from the SAP response. The events are shown in figure B.1, note that the notification is a subtype of the **APOSEventType** from the specialization project, meaning that it inherits the three failure classifier fields defined there. This is not the case for measurements.

LCI Nodes

The remaining logic is implemented in the **ExternalNodeManager**, which does not get the same benefit from the SDK. There is no good base class for creating this kind of custom behavior, so the solution is to create a full node manager by just implementing the **INodeManager** interface. An interface in C# just describes which public methods a class must implement. In this case that is one method for each service supported by the SDK.

The Browse service consists of multiple separate parts, as both LCI nodes and EqHub nodes can be meaningfully browsed. Each browse operation contains a filter, as discussed in chapter 3, and this filter can be used to determine which children of the node in question should be retrieved.

For LCI nodes, the TagID is always required, but relations and attributes can sometimes be skipped. The filter used is the following:
If HierarchicalReferences and the *Variable* NodeClass is requested then attributes are needed.
If *Object* or *ObjectType* is requested then relations are needed.
This simple solution does not consider every part of the filter, so the full filter needs to be applied to the result before it is sent to the client.

This can help with efficiency. Since the source systems are quite slow, limiting the resources

fetched for each query can speed up external applications accessing the server. After each request the result is cached in the **LCINode** class, so that the next time this node is requested the query is faster. This makes the server more responsive when doing complex operations. The full operation can be described with pseudo code:

```
For each requested LCI Node:
    If TagID has not been fetched:
        Get basic tag data from the database

    If this is an attribute
        and reverse references are requested:
        Add a reference to the parent node

    If the filter allows relations
        and relations have not been fetched:
        Read relations from the database

    If the filter allows attributes
        and attributes have not been fetched:
        Read attributes from the database

    For each resulting reference:
        If the item passes the full filter:
            Send it to the client
```

Note that if the node in question is an attribute, then browsing should return a reference to the parent node. This does not require reading from the source system, since the name of the node is encoded in the NodeId of the attribute, as described in section 5.2.

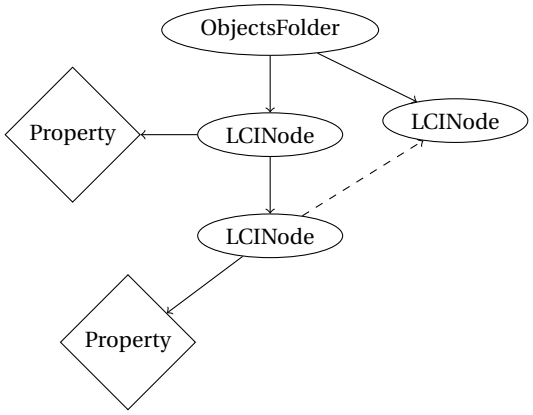


Figure 6.3: LCINode in hierarchy.

The LCI node in the OPC-UA node hierarchy can be seen in figure 6.3 (B.2). LCI Nodes are Objects, and the attributes are properties. Note the non-hierarchical reference shown as a dashed line in figure 6.3.

EqHub Nodes

LCI Node relations produce references to TEK Numbers, and TEK Numbers are both LCI tags and EqHub Equipment, so an **EqHubNode** contains references to both **LCINode** objects, and **Equipment** objects, as seen in figure 6.4.

It is not necessary to fetch attributes of the LCI Node, since these will at most just be duplicated from EqHub. The relations, however, are interesting, as they make it possible to reference all nodes in the LCI database that connect to a TEK number.

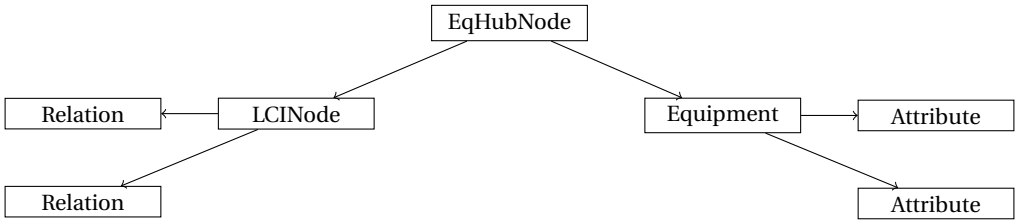


Figure 6.4: EqHub Node object structure. Each box is an object.

The procedure for browsing EqHub nodes is very similar to LCI nodes, and because the EqHubNode type incorporates an LCI Node, it can reuse much of the same functionality.

```
For each requested EqHub node:
    Fetch LCI references as needed, but never attributes

    If the filter allows attributes
        and the equipment has not been fetched:
        Read attributes from EqHub

    For each resulting attribute or relationship:
        If the item passes the full filter:
            Send it to the client
```

Using the method described in section 5.1 each EqHubNode is assigned to the APOS type hierarchy using attributes retrieved from EqHub. The structure final structure in context is seen in figure 6.5 (B.3), using a gas detector PIR7000 as example.

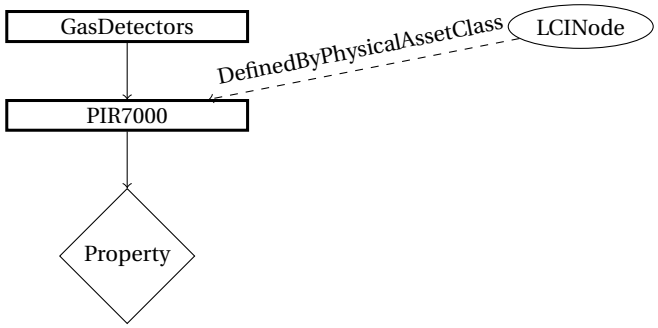


Figure 6.5: EqHub Node in hierarchy.

SAP Events

Since reading from SAP is so slow, and events are so rare, events are implemented as history only, which, while supported by the OPC-UA standard, is uncommon. UAExpert, for example, does not support just reading history without a subscription. Fortunately, this is simple enough to solve, since most of the event subscriptions are handled by the SDK. The node manager just needs to pretend to handle event subscriptions, and it will work fine for UAExpert.

The **Notification** and **Measurement** classes discussed above are used to construct OPC-UA EventStates. Event filters are applied to the event state by the SDK, producing a collection of attributes. The node manager only implements the code to produce a list of event states within the correct time range, given an LCI NodeId.

The events are cached for convenience, since reading them is so slow, so the final procedure is as follows:

```
For each LCI Node to read events for:
    If TagID has not been fetched:
        Get basic tag data from the database

    If the SAP platform code has not been fetched:
        Read SAP platform code from the LCI database

    If measurements have not been read:
        Read Measurements

    If notifications have not been read:
        Read Notifications

    For each resulting event:
        If the item passes the event filter:
            Send the requested fields to the client.
```

If the server requests a different type of HistoryRead, or requests to read from a node that is not an LCI Node, the server simply returns a bad status code. The resulting events can be seen in figure B.4 and figure B.5.

6.3 Structural Overview

Figure 6.6 shows a visualization of the data flow in the server, including the main data transfer objects. All the different resources require information from the LCI database, which is not a bad thing. It is positive that the LCI database contains all the inter-system context, since that means simplifies the system, and lets the LCINode object be the core of the node hierarchy.

This is also in line with the decisions made by Aker BP. The fact that so many systems are in use, that seemingly cover the same kind of information, is a consequence of Aker BP managing systems originally developed by multiple companies, with different policies and technology. LCI is being set up as a master system, and so using it as master in the server implementation is also the likely a good decision.

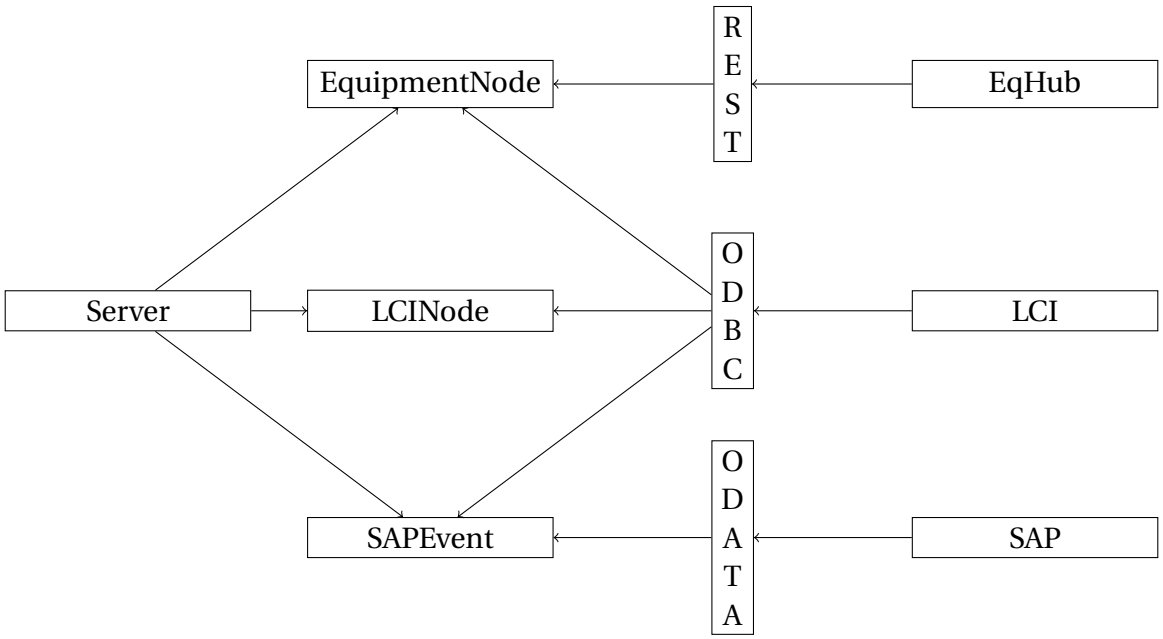


Figure 6.6: Data Flow in Server application.

Conflict resolution in a system like this would be very difficult. The way the server is designed means that each type of information is fetched from only one source. Instance information is fetched only from LCI, type information only from EqHub, and events only from SAP. This is a simplification, as LCI contains some type information, and SAP contains some instance information. The result, however, is that there is no room for conflicts, which helps ensure that this project is possible to complete.

Chapter 7

Testing the Model

The second half of this thesis is focused on verifying the implementation, and answering the three key questions from section 1.2. The first step is to ensure that the information model and the mapping described in chapter 5 holds when applied to a larger selection of equipment types. This will help verify that the modeling is sufficiently flexible, that EqHub and the LCI database contain sufficient information to map to APOS, and that the APOS information model is solid enough to allow for a consistent mapping, thus answering questions 1) and 3).

This chapter will discuss and implement two different methods of populating the contents of the server. Since the LCI system is the core of the server, the most intuitive way to populate the model is to simply search through the LCI database, and it is fairly trivial to extend the server to cover more tags. For now, there is a list of root tags specified in the code, and simply adding a few tags higher up in the hierarchy is sufficient to populate the hierarchy.

This, however, is not a good way to test the model, for several reasons. First, the LCI database is slow, and populating the hierarchy this way will mean retrieving the relations of thousands of nodes. At 0.5 seconds per node, 10 000 nodes takes an hour to populate, and the result does not really provide that much information, except verifying that all nodes in the LCI database fit with the model. Beyond that, there are enough nodes in the database that memory on the server might become an issue, since it caches the full hierarchy.

Secondly, this would mean discovering equipment types through LCI nodes, but most discovered equipment would not be gas detectors, meaning that most likely hundreds of new rules would have to be created, which would take a lot of manual work. Instead, it is better to fetch equipment types from EqHub, then find related equipment through the linked nodes.

So the test will involve retrieving all equipment of a given class from EqHub, fetching detailed information about each instrument, then attempting to add them to the hierarchy. The only manual work will be creating rules for mapping discovered types to the APOS equipment hierarchy.

7.1 Populating Gas Detectors

In order to do this, a new method is added to the EqHubSource class, which fetches all equipment with a given class. This is done using the `/Catalogues/[catId]/Teknos` endpoint, which allows specifying a “class” query parameter, to only retrieve equipment of a given class. There is

a maximum number of results returned each time, but it is possible to specify a "page" parameter to iterate through the contents of the database.

EqHub is external to Aker BP, and as such there is a number of equipment types defined in EqHub that have no associated tags in the LCI database. These should not be included, so part of the procedure must be querying the LCI database for relations matching the corresponding LCI tag for each TEK number. The full procedure is as follows:

```
Fetch all equipment types with a given class
For each equipment type:
    If there is no corresponding tag in the LCI database, skip
    If there are no relations for the tag in the LCI database, skip

    Fetch the full equipment type from EqHub

    Try to classify the equipment type, log information on failure

Save all relevant tek numbers
```

The code for this can be found in `Server/Program.cs/PopulateEqHub`. It is takes a class name as argument, which is first run with "Detector, Gas", in order to fetch all gas detectors from EqHub.

After running it once, the equipment types that were not classified are logged, and then used to manually expand the list of rules described in section 5.1. The remaining gas detector types defined in APOS, but not included in the specialization project also must be added. The new types are "H2SDetectors", "OilMistDetectors", "AcousticLeakDetectors", "O2DetectorsAndAnalyzers", "H2Detectors", "CODetectors" and "NH3Detectors".

This process can only be partially completed, as there are a number of issues:

- Some TEK numbers only have minimal information, or no information at all.
- Most equipment models are duplicated, and each version contains different information about the actual equipment model.
- Some types do not fit into APOS at all, like CO2Detectors. Part of this may be that APOS is focused on safety instrumented systems, while the LCI database and EqHub has no such restriction.
- Parts of the data model in EqHub is inconsistent. For example, some gas detectors specify "Gas type", and others "Type of gas".

The result can be seen in figure B.6.

Another issue discovered through this process is that a number of gas detectors lack the "Model" field, which was originally used as DisplayName in OPC-UA. In order to avoid blank DisplayNames, which is not ideal, the TEK number is used as a fallback.

7.2 Populating PSVs

This algorithm can easily be reused with another equipment type, to verify the ability of the model to extend to different equipment types. The PopulateEqHub method is called with "Valve, Relief", in order to populate the APOS L2 group "PressureSafetyValves" (PSV).

This illustrates that while some types, like Gas detectors, require complex mapping from EqHub to APOS, others correspond much more easily. In this case, the "Valve, Relief" class in EqHub matches the relief valve L2 group, which is under the "PSVs and rupture discs" L1 group, perfectly. A single rule is sufficient to map these:

```
- name: PressureReliefValves
  fields:
      Class: ["Valve, Relief"]
```

This works very well, however it does take a while, as there are close to 2000 different types of PSVs in EqHub, 500 of which are related to something in the LCI database. In order to do this on a larger scale, it might be reasonable to modify the queries to the LCI database to be able to read relations of multiple nodes at the same time, which is possible with SQL. Some of the result can be seen in figure B.7.

7.3 Results

This process highlights a few facts about the implementation and the concept as a whole:

First, the configuration schema is flexible enough and works very well. This is essential for a project like this, which involves a great deal of mapping: a flexible configuration schema. Any mapping process such as this will inevitably involve some amount of manual work. By using a filter/pattern based mapping such as this one, rather than a manual one per equipment type, one can define an increasingly accurate "pattern". Most of the gas detector types that were correctly mapped in the end were never manually looked at, they were just similar enough to other types to be affected by a rule made to match a different piece of equipment.

This is a form of model based learning, it could even be seen as a form of primitive artificial intelligence. The program is taught a set of individually simple rules, that combined define a complex model.

Second, while the EqHub API is fast and fairly flexible, the actual contents are often lacking. This is unfortunately not unexpected in a system like this. The data is entered by a number of different companies, and a larger number of different people, over extended periods of time. The lack of a clear data model for most fields (Class is a notable exception), means that multiple rules are needed to cover the same types, i.e. the "Gas type" and "Type of gas" issue described in section 7.1, and the lack of data for many types mean that good classification is difficult.

Finally, the types are numerous enough that pre-fetching everything is too slow, and this also creates the issue that the server does not reflect the hierarchy efficiently. Instead, the server could be modified to save the category of each TEK number, then fetch the contents on demand, instead of saving the TEK number itself and fetching attributes on startup. For now, the current solution is sufficient for this project. Even fetching the 600 different equipment types currently registered from EqHub does not take more than a few seconds, as expected of a modern API.

These results can be used to propose changes to the server and source systems later.

Chapter 8

Usage

The purpose of creating an OPC-UA server to collect data is to provide a common interface to external programs. In order to explore the potential value of a proper server implementation, this chapter investigates a few different ways to use the server, and how a common OPC-UA platform can be used to provide value to a company like Aker BP.

This also serves the purpose of verifying that the server is capable of representing the data in a way that is usable by Aker BP and the APOS project, which helps with answering question 2) in section 1.2, as well as testing how well the source system APIs handle the larger load associated with practical applications, answering question 3).

This thesis looks at three concrete ways to use the server. The first is the external tool UAExpert, used to generate figures as seen in appendix B. This tool required no configuration beyond entering the IP address of the server, and without any knowledge of the underlying systems, it was able to explore and display the data in context.

Figure 8.1 shows the use of a general OPC-UA platform, to connect external tools to the OPC-UA server. This chapter looks at two such tools. A custom application developed for using the OPC-UA SDK to analyze data collected by the server, and an external application developed by Aker BP partner Cognite.

8.1 General Uses

In general, there are three main benefits of a server like this:

1) It has contextualized data across systems.

2) It allows access to the data over a single unified protocol.

3) The OPC-UA protocol is standardized and widely adopted.

None of the existing applications discussed in chapter 4 have all these advantages. The Aveva LCI app has some cross-system context: it links to the EqHub page, and incorporates some limited data from SAP. The OPC-UA server expands on this by providing the EqHub data directly, and retrieving Notifications and Measurements from SAP.

2) and 3) were not provided in any real extent by any of the three systems. ODATA can in theory be used as a general API, but the ODATA API currently available in SAP is not sufficiently

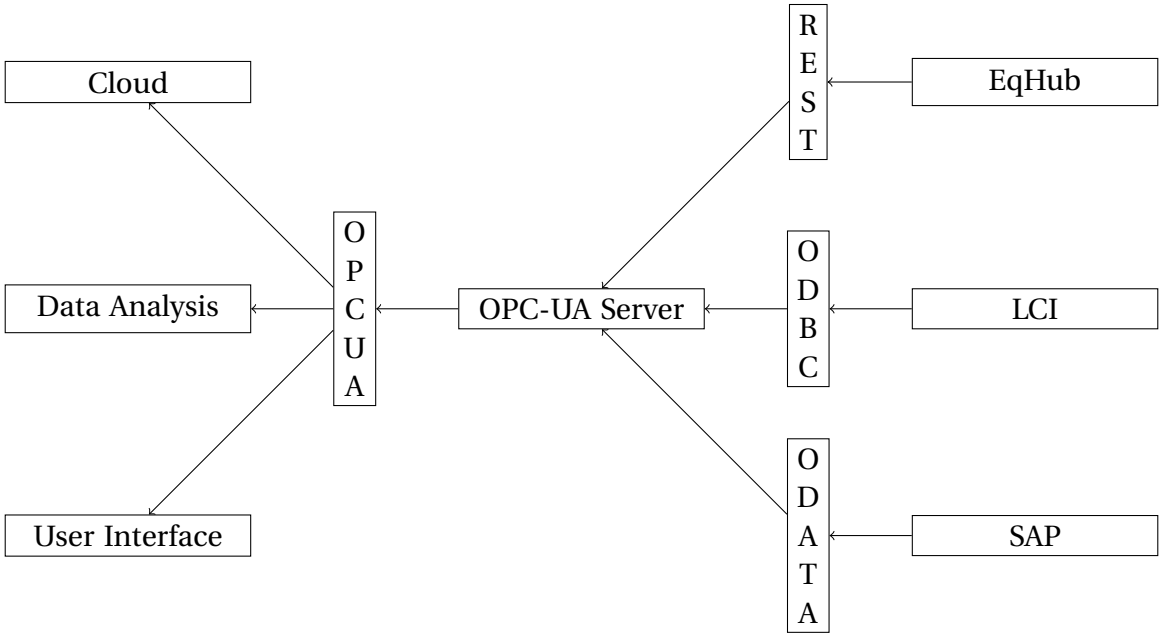


Figure 8.1: The server provides an interface to the underlying systems

developed to be used in such a way. Both the LCI database and EqHub have useable, externally accessible APIs, but using either one requires external context. The LCI database requires AttributeName maps, and an understanding of the different tables, and EqHub requires API documentation.

The OPC-UA server does not have this disadvantage, at least not to the same degree. Parts of the logic, such as reading events and attributes of equipment instances only requires an understanding of the base OPC-UA information model, and understanding the equipment hierarchy only requires an understanding of the ISA-95 companion standard.

An application capable of exploring OPC-UA servers in general is, using the server, able to access parts of LCI, SAP and EqHub. This, in the end, is the primary reason for creating something like this. If the goal was just performing a single operation, or making a single cross-platform query, developing a full OPC-UA server would be excessive. Simply creating a single application using the code for accessing the source systems would be much easier.

The advantage of creating the server is that *multiple* applications can now access it, including applications without any native support for the source systems. It allows integrating with the existing OPC-UA ecosystem, meaning that third party systems that support reading from OPC-UA could use the server as a gateway to Aker BP data.

In order to construct such a gateway without an interface like this, the third party system would need to support all three source systems. It may be possible that some of these support SAP, but it is extremely unlikely that they support LCI in a good enough way to provide any useful information without a lot of further processing, and EqHub is almost impossible, seeing as the API is custom-built and fairly new.

8.2 Calculating Failure Rates

The first application covered here is a small OPC-UA client app that gathers statistics on failure rates from equipment types, developed as part of this project. Companies such as Aker BP are required by law to provide a yearly report of DU (Dangerous Undetected) failure rates for specific groups of equipment. One of these is fire and gas. The procedure is simple enough:

- Once a year, perform a test-reading of each fire and gas detector. These are registered as MeasurementDocuments in SAP.
- At the start of the next year, look up measurements of all tags with the specified object types in SAP, created in the last year.
- Calculate the failure rate by dividing the number of failures by the total number of tests.

A similar process can be created for a client accessing the OPC-UA server:

- Browse the GasDetector and FireDetector hierarchies for all referenced LCI tags.
- Read event history for the last year for all referenced tags.
- Calculate the failure rate by dividing the number of failures by the total number of tests.

The code is included in the *FRReader* folder, which is the third and final C# project included with this thesis. See appendix A for details.

Browsing the gas detector hierarchy returns 1800 tags. Despite reading measurements for 20 tags per request, retrieving all measurement documents took close to an hour, mostly due to the slow speed of the SAP API. It produced 1234 measurements, of which 3 had a non-pass status, resulting in a DU failure rate of 0.24% per year. Performing the procedure in SAP takes a few seconds and produces around 2700 documents, highlighting the incompleteness of the information model, and the inefficiency of the SAP API.

This highlights a few clear issues with this approach:

- Many tags lack TEK numbers, or the TEK numbers have wrong classes in EqHub.
- The SAP API is extremely slow, despite it being clear that the underlying SAP system is fast.
- This requires all three source systems to be correct, while the pure SAP approach requires only SAP to have the correct tag types.

The advantage of doing it this way would be that it could produce failure rates per TEK number, which would be much more difficult to do in SAP. Using this one could compare the failure rates of point gas detectors versus line gas detectors, or identify equipment models which performs worse than expected. Of course, this still would not produce ideal results, since EqHub contains so many duplicates of equipment models.

This is, in general, an example of how APOS would use this model. One of the focuses of the APOS project is how the information model can be used to analyze equipment and compare it to requirements for each object type, and hence identify whether tests need to be performed

more or less often, if equipment should be replaced, etc. The processes described by APOS can be implemented using this OPC-UA server. In fact, it should be possible to create an OPC-UA client for APOS based only on the information model designed in the specialization project report, then apply it to implementations such as this one.

The failure rate calculator can be seen as an example of an *APOS context application*, an application that is aware of the APOS information model, but not the particular implementation.

Based on the results seen here, it is safe to conclude that the current approach is insufficient to handle this task. Not only is it several thousand times slower than doing the same through SAP, but it produces less than half the number of results. In part, this is due to the approach, which relies on consistent data across three different systems, where the SAP approach only requires SAP to be consistent. It also potentially provides some numbers indicating the degree of consistency between the three systems. Ideally, both for this project and Aker BP in general, the two processes should always produce the same results.

8.3 Mass Data Extraction

The second main use of the server is to access it with context agnostic applications, meaning applications that have no understanding of the contents beyond that they are OPC-UA. UAExpert is such a tool. Extracting data to Cognite Data Fusion (CDF), a data platform developed by Cognite[9], is relevant to Aker BP. Cognite is backed by Aker BP, and performs the role of storing data in the cloud, in order to make it accessible, as well as transforming and analyzing the data to give it value.

In a way CDF might be considered an alternative to this project. Instead of trying to dynamically contextualize data from the source systems in real time, Cognite creates transformations and rules that contextualize data later, which in the end is more flexible. CDF also duplicates the data, so that even if the source systems are slow, the data can be fetched later. That said, CDF inevitably involves some delay, since it is reliant on extraction, rather than simply providing an interface to the original copy of the data. OPC-UA, by contrast, can in principle be a pure software layer that does not duplicate any information, allowing it to be used for real-time applications.

On the other hand, the two systems are not necessarily competing, since due to the existence of the Cognite OPC-UA Extractor, development of an OPC-UA server also opens the possibility of inserting the data into CDF with minimal further work, as this section demonstrates.

8.4 CDF

The CDF data model is fairly simple. While there are a number of resource types[10], the following are the relevant ones for this project:

- *Assets* are *Object* equivalents. Like OPC-UA objects they usually represent physical or logical units, and are used to connect and contextualize data.
- *Time series* are *Variable* equivalents. Unlike variables, they effectively only store historical data, without a concept of “current value”, other than just the last point in the series.

- *Events* are, of course, OPC-UA *Event* equivalents.
- *Relationships* are *Reference* equivalents. CDF also defines a separate system for parent/child relationships between assets and time series, so relationships can perhaps be seen as more of an equivalent of non-hierarchical relationships in OPC-UA.

In order to fully use the tools Cognite provide, the data has to be inserted into CDF, and it has to be contextualized. Of course, the more context the source system can provide, the easier it is to contextualize after the fact. Among the tools Cognite has created are "Extractors"; tools that read data from an external system and write it to CDF using a REST API. One of these extract from general OPC-UA servers.

OPC-UA Extraction

Running the extractor against the server requires a little configuration. Most servers are much faster than this one, and so the extractor will be modified to read a limited number of nodes at a time. It must also be made to extract the full server hierarchy, so that it extracts both types and instances.

Using this configuration, which is included in `cdf-config.yml`, the extractor is run on 21 LCI tags, and they are replicated to CDF. Giving the following results:

- Figure B.8 shows part of the LCI node hierarchy. This is through the Cognite app "Fusion", which can be used to explore the various resources. These are all assets, as OPC-UA objects are mapped to assets in CDF.
- Figure B.9 shows the metadata of one of the LCI nodes. Properties in OPC-UA are mapped to metadata, a collection of key/value pairs on each asset, time series or event in CDF.
- Figure B.10 shows metadata and hierarchy for the gas detector type all the LCI tags belong to, PIR7000. Since the properties of ISA-95 types are not of the standard OPC-UA property type, but ISA95ClassPropertyType instead, this required manually configuring the extractor so that nodes with TypeDefinition of "i=4885" were treated as metadata instead of as time series.
- Finally, figure B.11 shows a single measurement event. The extractor has automatically detected that some nodes expose event history, and has extracted events from all of these.

The ability to extract data like this is another reason why a unified OPC-UA platform is valuable. It not only provides a tool for internal use, but it also allows integration with external systems that support OPC-UA, and by extension, it allows all this data to be used through the various applications provided by Cognite.

This kind of "Eco-System" based thinking is very valid when thinking about modern industrial data. Providing a sufficiently general interface can essentially integrate Aker BPs internal systems with a much larger eco-system, and all tools that are developed for that eco-system can then be used to access Aker BP data. This effect is also self-reinforcing. The larger an eco-system becomes, the more companies invest in connecting to it, and the more valuable it becomes.

OPC-UA has an important role in this development, as its self-describing nature means that integrating it into an eco-system can often be done automatically, using tools similar to the Cognite OPC-UA Extractor. In a perfect world, every system would be accessible using generic interfaces in such a way that all tools would be able to access data from any system.

The Cognite OPC-UA Extractor is invasive and extensive. The fact that it is capable of finishing its run on the server developed here indicates that the server is stable, and that there are no unforeseen issues associated with rapid extraction from the source systems. Answering in part question 3) from section 1.2, in that while the source systems are quite slow, it is possible to intelligently throttle the load in order to read large amounts of information.

Chapter 9

Limitations and Extensions

Chapters 7 and 8 investigate the three key questions from section 1.2. Using these results, this chapter will discuss the issues in the source systems, the server and the information model, and what could be done to solve or mitigate them, as well as possible extensions to the server.

Sections 9.1, 9.2 and 9.3 discuss each of the source systems in turn, to answer question 3) in detail, and suggest how the source systems or the approach could be modified to solve these issues, and section 9.4 explores how an ideal solution might look. Section 9.5 summarizes the limitations of the information model encountered in chapters 7 and 8, and attempts to answer questions 1) and 2). Finally, section 9.6 lists the known limitations of the current server implementation, and how it could potentially be expanded in the future.

9.1 LCI

The core of the Aker BP solution is the LCI database, and similarly it is also the core of the OPC-UA interface. The reason for this is most likely just that it has been designed especially for Aker BP, and it is flexible enough to do the job. However, it seems that the reporting database is not intended as a way to access the data automatically, but as indicated by the name, for "Reporting", i.e. a systematic way to dump and query data.

Instead of creating a front end application, they have chosen to create a few convenient views. There is nothing inherently wrong in this approach, SQL is far more flexible than any frontend dumping tool is likely to be, however the exposed database is not very fast, and not easy to work with due to its difficult structure.

Most of the database schema has not been made available here, and according to sources in Aker BP this is considered proprietary code by Aveva. Fortunately, even if details are unavailable, the database tables are exposed, and so it is possible to at least see which rows exist and what their data types are, which can be used to guess how the database is built.

The information model is quite primitive, meaning that the actual information model is not built as part of the database, which is how this is usually done, but most likely exists only in external documents or code. This is illustrated by the issue discussed in section 4.2, that attributes are classified by numeric "names". Without the external context it would be impossible to understand the contents of the database.

There are several examples of this. First, the associations table contains associations from a

"source" to a "target", where both can belong to a variety of different tables. This is problematic in database design, since it means that there is no singular definition of each column. Take the "SourceId" column. It may refer to a tag, a document, or something else entirely. A more normalized database schema would split this up, so that there was one table for tag to tag relations, one for tag to document, and so on.

Another important concept that appears to be lacking is the idea of "Foreign Keys", which are essentially ties between tables in the database. This is just not possible for the associations table, since each column reflects data in multiple foreign tables. The "TagId" in the AttributesOfTags table should also be a foreign key. This does not appear to be the case, unless it is part of some hidden schema.

Combined these issues result in slow load times. Most likely, at least some of this performance hit is caused by the poor structure. Whether this really is a copy of the production database, or some transformation, is not clear. It is also possible that the reporting database is lacking computational resources that are available to the live database.

Due to the lack of documentation, coming to any solid conclusions about the database is difficult. The schema is simple, and flexible, but understanding the contents requires external context, which is not easily accessible. With increased automation, the lack of a way to access this information automatically and on-demand is almost certainly going to become a problem. Speed and ease of access is *necessary* for centralizing and consolidating data from different systems.

If the LCI database is going to be used as the basis of an OPC-UA server like the one described in this project, it will need to be improved in some way, if only to make the current solution faster. Fortunately, there are a few steps that could in theory be taken:

- Ensure that the table is designed optimally. Indices and foreign keys can help performance, and should be added wherever it would help. An effort should be made to make the database efficient to access for external systems.
- Add the necessary contextual information to the database. In particular, the names of attributes, the valid attributes for each tag class, and the valid associations between classes. This information needs to be encoded in the database. Either in the model itself, or in a separate table.
- Upgrade the server the database is running on. Databases are usually very good at using extra resources, upgrading the server itself would allow the data to be read in reasonable time.
- Obtain the database schema from Aveva, if Aker BP does not already have access to it. Aker BP should in general require external software companies to provide documentation not just about the frontend applications, but about backend APIs.

In practice, however, the database itself is not very complicated, and the best solution might be to replace it with something more in the line with the needs of Aker BP. Based on the limitations outlined above, a few key requirements can be posed to a theoretical replacement:

- It should use modern technology. The amount of information provided from each instrument, as well as the number of devices, is growing quickly. A solution should be built to

handle many times the current load, and it should use scalable software that will not have the same issues in another ten years.

- It must have a solid API. An SQL based interface is perfectly fine for something like this, but it must be built with external access in mind.
- It should use a stricter information model. There is a large field of theory regarding the normalization of databases that should be considered when developing a relational database, if that is the chosen data model. At a minimum, the database should not rely on large amounts of external context, and should not contain huge numbers of empty fields.

Rather than making the database capable of modeling any future content with a minimum of tables, like the current solution, the replacement should seek to model *only* what Aker BP needs now, and instead leave room for modifying the information model itself in the future, if necessary. While flexibility is usually a good thing, it is more important that a model is intuitive and efficient.

In fact, Aker BP is already in an early stage of designing a potential replacement for the LCI database. Hopefully the findings in this thesis can be helpful in this process. A redesign is a good time to ensure that the system is future proof, and a future proof LCI system will almost certainly include a solid, efficient external API.

9.2 SAP

SAP is designed to be an all-in-one solution for industrial management, which is not quite in line with the core concepts of this project, using flexible interfaces to connect different systems. The general idea is that a company will use SAP for all asset management. The ODATA solutions developed by SAP are most likely not really meant to let other systems include SAP data, but to provide it to user applications such as Microsoft PowerBI. This does potentially make this project more complicated, but the official ODATA API described in section 4.4, or the current ODATA APIs in development at Aker BP would likely have been sufficient.

Previous projects to extract data from SAP in Aker BP have apparently used direct access to underlying databases, which has the disadvantage of lacking the processing that is usually done by SAP. It is possible that this project could have used a similar approach, but that would have required further access and information about SAP, and this possibility was not explored further due to limited time.

It is difficult to make any other conclusions about the SAP system. Control- and diagnostic systems only actually produce a limited subset of the failure events generated by equipment, those that were detected automatically. As mentioned, the relevant events for safety analysis are those that are DU (Dangerous Undetected), which are typically only generated by operators performing function tests. As a consequence, much of the data relevant to APOS is in SAP, and reading from there is essential for this project, yet the chosen solution is not acceptable for a final iteration of the server. The alternatives discussed here are:

- The official SAP ODATA API could be added to the SAP instance running at Aker BP.
- The Aker BP ODATA API could be expanded to cover the necessary data for this project.

- The server could directly access underlying databases.

There might also be other alternatives that this project has not explored, like a different API for SAP. It seems like it may be possible to invoke SAP transactions directly from the command line, which could be an alternative. What this project has shown is that connect to SAP with external applications is not a simple task, and that SAP integration is likely to be a challenge in the future of Aker BP.

9.3 EqHub

EqHub has the advantage over the other two source systems in that the API is fast, efficient, flexible and feature rich enough to be sufficient for a project like this. Even reading several hundred pieces of equipment from the API takes less than a minute.

The idea of EqHub is a good one. Gathering this kind of data in a central location and making it accessible greatly improves the possibility of modeling. As discussed at length in the specialization project, avoiding duplicated data is an essential part of a good information mode, and EqHub effectively applies this to data shared between companies. Ideally there would exist a single, universal system, using a single unified information model, which contained *all* type information.

EqHub can be seen as a sort of prototype of something like that. What it does have is information that is centrally stored, supplied by a number of equipment manufacturers, and easily accessible through a solid API. The API works well enough for this project, but it too could be improved. In particular, the fact that the equipment retrieved by filtering based on class only contains minimal information about each TEK number means that each individual TEK number must be fetched with a request each in order to further classify large numbers of equipment.

This has to be done with individual requests for each TEK number, it would be even faster if the Sharecat API had bulk request endpoints, to fetch many items using a single request. The backend is likely fast enough that bulk requests would be orders of magnitude faster, in theory making the storing of TEK numbers completely unnecessary. If bulk fetching was possible, a search filter could be assigned to each L1 or L2 equipment group. When browsing a group, these filters would be used to fetch an initial collection of equipment, then enough data to classify them could be fetched with a bulk request, removing the need for buffering in the server.

This is a minor issue, however. What EqHub really lacks is a consistent information model. The value of the data to automated systems is greatly reduced if it is inconsistent, and while there seems to be some system to data in EqHub, it is not strictly enforced, and it appears to be largely up to the users how the data should be organized.

Of the three source systems discussed in this thesis, EqHub is the one that would have the greatest benefit from an equipment model like the one presented by APOS. The APOS equipment hierarchy is detailed, extensive, and backed by industrial standards. Ideally, EqHub would not just use, but *enforce* use of a model like APOS to ensure that the data is consistent.

As it is now, EqHub can provide useful information to users, and *some* useful information for automation. The API is sufficient for production use, although it could use some better bulking, but the actual contents could use some work. The following measures could be taken, in increasing order of complexity:

- Enforce the presence of some fields, like "Model". All equipment has a model, or something that can reasonably be used as a model name, some fields will *always* be defined, and there is no reason to allow them to remain empty. The amount of manual work this creates per equipment type is minimal, and it greatly enhances the model.
- Enforce unique model/manufacturer pairs as far as that is possible. A single instrument model should not be represented by a dozen different TEK numbers.
- Plan the possible attributes per equipment class in greater detail, so that there is a limited list of legal attributes for each class. Require these to be specified.
- Enforce a limited set of legal values per attribute. This would have to be based on an extensive model, ideally there would be no "other" option. Users creating new equipment would be presented with a form for filling out the properties of the new model, where the values of each field is selected from a limited list of legal values.

The last two items could in theory be based on APOS. The requirement this poses for the APOS project is producing a versioned, machine-readable, complete document detailing the legal values and attributes of each class. Ideally such a document would be provided online, so that sources like EqHub could retrieve an updated version automatically. It would also mean that APOS, or a similar project, would need to be expanded to cover non safety-critical equipment groups, or it would not be feasible to apply it to EqHub.

This solution would pose a higher requirement on equipment providers, but considering the already considerable requirement posed for equipment documentation, and the value this provides, it is a minimal burden. The number of new equipment models produced per year, and the amount of time it takes to create formal documents, test and verify the contents, and so on, is orders of magnitude larger than the time it would take for a manufacturer to fill this data into EqHub.

It is much easier to push a system like EqHub to provide the translation from manufacturer specific information models to something like APOS, rather than relying on *every* manufacturer to use the same model.

In conclusion, EqHub is a great concept, and the API is a very solid start, but the current contents are not consistent enough to fully rely on for automation, at least for the data referenced from Aker BP.

9.4 Restructuring

An interesting exercise to better understand the current state of the system is to explore what it would take to replace the current system with something completely different.

As discussed above, the LCI system appears to be overtaxed, meaning that it represents a much more complex information model than it was originally intended to. The TEK numbers being tags is an irregularity, in that it means that the "Tags" table now contains mainly tags representing assets, and a small list of TEK Numbers which do not. The current information model used by the LCI database could be used to represent *any other* information model. Instead, the model should be designed for the specific purposes needed by Aker BP.

As stated, the idea of EqHub is good, and an ideal solution would keep it, or something similar. As it stands, the LCI database contains some copied information from EqHub, but since the API for EqHub is flexible and efficient, this is largely pointless, and an improved system would drop this duplication entirely. Ideally, the LCI replacement would integrate tightly with EqHub, to make duplicate TEK numbers and inconsistent data less likely.

Finally, SAP. SAP is the most difficult of the source systems to modify. It would be fairly trivial to transfer the LCI data model to a more specialized data model, but SAP overlaps in part with both EqHub and the LCI database, while also having its own tag system and internal structure.

The issue is that SAP is designed to be a monolith, without really interacting with external systems. SAP ideally is meant to manage *all* the data: equipment requisition, measurements, status notifications, personnel, tags, and so on. At the same time, it is fairly rigid in the way it wants to represent this data, with platform numbers and codes, as well as tag names. Other, external systems that want to interact with SAP needs to translate and conform to the SAP model if they want to retrieve the data.

As a consequence, it is almost impossible to entirely replace SAP. An approach might be to take specific parts of SAP and move them to a different system, or somehow modify the current SAP setup to integrate better with an external model. As it stands, some data in SAP replicate or conflict with data stored in the LCI database or EqHub. SAP functional location is similar to LCI tag attributes, and SAP construction types is similar to EqHub equipment. These parts could be removed from SAP, to reduce the number of opportunities for conflicts.

Suggested Replacement

Ignoring what is or is not possible for now, it is useful to visualize the ideal industrial system, to understand how the current system can be modified to get closer to this ideal. The suggested system consists of two main components: First, a database for tags, similar to the current LCI database, which contains information from the control system, manual reporting like functional tests and notifications, a collection of tag specific documents, and a simple reference to a TEK number in EqHub.

Secondly, EqHub would remain and contain what it contains today, but organized and expanded as described in section 9.3.

The suggestion uses a modern approach called an "Asset Administration Shell" (AAS)[40]. The idea is to collect the information in a "Shell" around each asset, so that all instance data is accessible in the same place, conceptually close to the actual asset. A central system like the LCI database is now would then have the much more limited role of just connecting and referencing the shell of each asset. Some external system would contain the connections between each asset, but any further data storage would be delegated to the shell.

The resulting model is very clean, see figure 9.1. In order to make this practical, it would most likely mean somehow integrating this with SAP. It might be possible to reference these shells from SAP in some way. Perhaps an effort should be made to investigate how SAP can be modified to fit with a modern approach to information modeling, such as the Asset Administration Shell. If not, then at some point it may need to be replaced, at least in part.

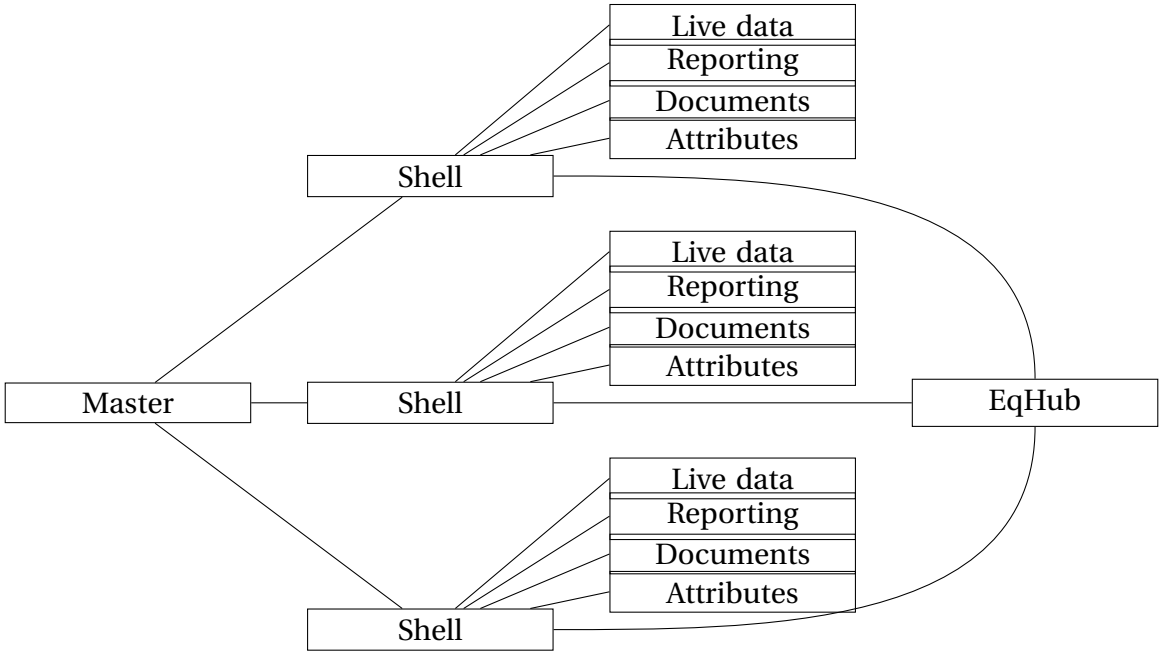


Figure 9.1: Suggested AAS-based structure.

9.5 Information Model

While the model introduced by APOS is solid, it is worthwhile to discuss whether the issues uncovered through testing and using the server are due to limitations in the information model, and not in the source systems or the server.

Limited Scope

APOS intentionally only covers equipment related to safety instrumented systems. While this is a deliberate choice, it does create some serious issues. For a company like Aker BP it is not ideal to apply a complex type model to only a small part of their system, which does not even cover the entirety of their safety-critical equipment, while keeping a different model for everything else. If APOS is to be used in practical, technical solutions, and move beyond just being used for modeling in smaller projects, it *must* cover all types of equipment.

One solution to this is to make the information model expandable, so that others can extend it to cover other types of equipment, but it is worth noting that the limited scope of the project makes it a less likely candidate for a standard information model for equipment. The general idea of a model like APOS applied to all types of equipment is good, but as it stands using APOS in practice would either mean applying it to part of the information, and using a different information model for the rest. Alternatively, equipment related to safety instrumented systems could be separated into another system entirely.

Both solutions encounter the problem of having to connect the two systems, and teach operators how to use both. Discussions with both Aker BP and other companies partnered with the APOS project indicates that while there is general interest in a common information model, the fact that it can only be applied to a subset is potentially problematic.

Model Inflexibility

One notable difference between the classes in EqHub and the L2 groups in APOS is that L2 groups, in many cases, are more specific. This highlights a problem with the APOS model that is difficult to solve, and that reflects a general challenge in this kind of type-based modeling.

In ISA-95, the concepts of *Equipment* and *Physical Assets* are different. Equipment class refers to the functionality of equipment, while Physical Asset refers to its make and model. In the chosen interpretation of the APOS model, L3 equipment is chosen to correspond to Physical Assets. In a way, this means that the Equipment class of a node is defined by the L2 group its physical asset type, but this immediately highlights an issue with the model.

The reason why this is not the way ISA-95 models functionality is that some physical assets can potentially perform multiple separate functions. If APOS is general enough that an L3 equipment type could belong to multiple L2 groups, then there is no consistent way to model it in the current model. For example, it is not entirely unfeasible that a gas detector could be made that is capable of being configured to use both line-based and point-based measurement, which would place it in two separate L2 categories.

In ISA-95 this would be solved by simply making the PhysicalAsset class the actual model of gas detector, and using the Equipment class to indicate which way it is being used. One way to solve this issue might be to simply create two different L3 nodes, one for each function of the gas detector, but this too is not ideal. Perhaps in practice ISA-95 needs to be reconsidered entirely, and be replaced with some other OPC-UA companion standard capable of maintaining the APOS hierarchy without creating this issue.

Alternatively, the issue might be seen as a criticism of the APOS hierarchy, in that some categories may be too specific. More general categories would be less likely to have this issue. Either way, APOS should address this issue, and make a decision on how to handle types that fit into multiple categories. As technology advances, this is likely to become more common, even in safety instrumented systems.

9.6 Further work on the server

The server is little more than proof of concept, which is what it was intended to be. There are several concrete steps that can be taken to turn it into a server that could be used in practice, or integrated with the unified OPC-UA platform in development at Aker BP.

Some decisions were taken due to performance limitations. These will be ignored for this section, as it can be assumed that a fully functional solution would have taken the necessary steps to resolve the issues using the methods indicated thus far in this chapter.

As such, there are two main fields that should be addressed. First, the server is not fully compliant with the OPC-UA standard, due to some shortcuts made to save time. Next, there is more relevant data in other Aker BP systems that can and should be integrated with the current model.

Compliance

In order to ensure compliance with the OPC-UA standard, a few steps must be taken. The SDK handles a large part of this challenge, by dealing with the most difficult part of the OPC-UA

standard to implement: communication, encoding, security and user sessions. There are two requirements missing from the Browse service implementation.

- It does not respect the BrowseResultMask flag, meaning that all fields are returned for each reference, no matter what flags are set. This is not a big problem, clients are unlikely to run into issues because of this, but it is worthwhile and fairly simple to fix.
- It does not respect limits, and does not provide ContinuationPoints. There are two steps to fixing this. When browsing, fetch only as many results from the source system as requested by the user. If this is not possible, for example if all attributes must be fetched at the same time, then only the requested number of references should be returned, along with a continuation point, and the remaining references should be stored until Browse is called again with the continuation point from before.

The server should in general change to no longer buffer data beyond what is requested through continuation points. This is discussed at length in chapter 3, but an ideal OPC-UA server has no internal state. The ContinuationPoints are a deliberate exception to this, but these are generally temporary, limited in number, and the client has some measure of control over them.

Buffering is not acceptable for the final iteration of this server, as the full tag hierarchy would easily fill gigabytes of memory. If the server is to serve as an interface, then this buffering must be eliminated, even if this results in superfluous requests to the source systems due to imprecise APIs. Making this change is not very difficult, but it would require the source systems to be able to handle the traffic better than they do today.

Extending the server

As mentioned, this project interacts with two separate but related projects. APOS, and the Aker BP project to create a universal OPC-UA platform. A natural next step would be to look at integrating this server with the existing OPC-UA servers for the various control systems. These are not structured based on the LCI tag structure, but rather based on internals of the control system. In theory, the server could be expanded to read from these tags, to obtain live values for the control system configuration and physical measurements.

This would not be trivial, however. The data present in these servers is typically organized not based on tags, but rather based on the control system itself. Each tag may have several values, for control inputs, outputs, error values, and so on. The data in the control systems will typically not be organized in a way convenient for contextualizing in the data model. Integrating this data is a considerable project.

Of course, once contextualized, this server is designed to be easily accessible. Combining data from the control systems with the rich context from LCI and EqHub and the reporting from SAP would be an important step towards creating a unified OPC-UA platform.

The extended model could be as seen in 9.2. A new node type, which contains the inputs/outputs on a standard format as well as configuration and other properties relevant for the control system in question, would be added. This structure could be quite complicated, and would contain references to other objects in order to model control loops, or other complex information available in the control system.

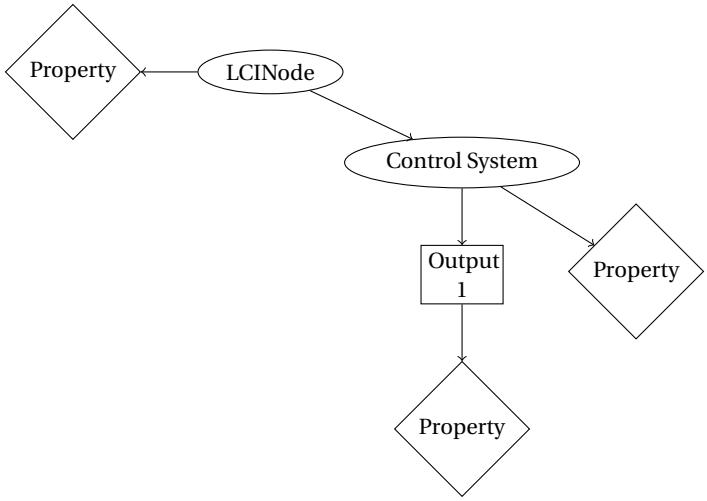


Figure 9.2: Expanded LCINode.

Figure 9.3 shows the extended information flow. Most likely, contextualizing the control nodes would require information from LCI or SAP, which would have to be fetched. The server would likely maintain one connection to the control system OPC-UA server for each connecting user, and call OPC-UA services on those underlying systems for the relevant nodes.

Path to Production

Given this it is relevant to ask, what are the concrete steps to put this server into production, deploying it and using it as an official solution?

Assuming the general mapping from source systems to APOS is acceptable, the first step would be to limit which attributes and fields are read from the source systems. Many properties seen thus far are likely to not be very useful. Beyond this, APOS aims to define strict information models for each type. The data from the source systems should be mapped to this information model, in order to reflect that part of APOS.

As discussed in the specialization project, ideally every field would be limited by a strict data-type limiting the legal values for each attribute. Similarly, the legal fields for each equipment type would be strictly limited, and any that cannot be mapped over would be ignored.

Any holes in compliance mentioned thus far would need to be filled, and the issues mentioned with respect to the performance of the source systems would need to be solved, either by finding different APIs, or upgrading the backend systems the server currently uses.

Moving on, the mapping would need to be filled out for all equipment and failure classifier types. In practice, in order to use this the way Aker BP wants to, the APOS model itself would likely need to be extended with equipment not related to safety instrumented systems.

Next would be integrating this by now stable model into the other information, as discussed above. A server which places data from control systems in the LCI context, and is capable of accessing data as an interface, without replicating, could be called “version 1”. Beyond that, the server would eventually be expanded and combined with other projects to cover all tag-related data in Aker BPs systems.

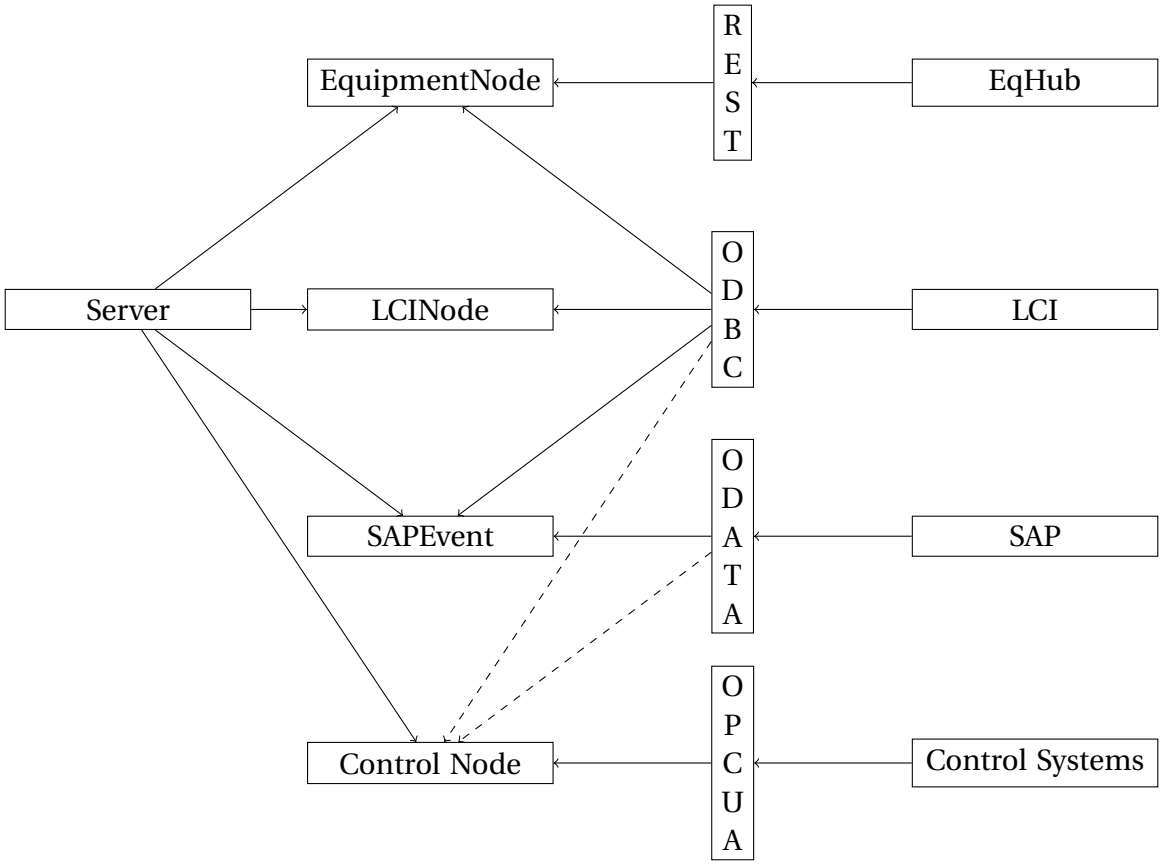


Figure 9.3: Extended data flow.

Chapter 10

Conclusions and Discussion

10.1 Summary and Conclusions

The goal of this project was to investigate the challenges of applying the APOS OPC-UA model to a set of existing source systems. The result stretches the limit of what was achievable, given limited time, and the source systems in question. All three systems had factors that complicated the development process.

The LCI Aveva information model, while functional, is very general. The type of information model used by the LCI database can in theory be used to implement any other information model, meaning that the information model itself does not describe the contents to any meaningful level. External context which describes the contents is needed. In addition, the database appears to be inefficiently constructed. It was necessary to bypass the provided views entirely, as they were too inefficient. Even with optimizations, it takes 1-2 seconds to fetch each node, and while parallelization might be possible, it can only help a limited amount if the issues are caused by the database or hardware constraints.

The SAP API also has performance issues. While the backend appears to be more than fast enough, there is no currently active and accessible API except for the very slow one used for the Fiori app. While the app works fine, the API is difficult to use and generally unsuited to automation. Parallelization is possible, but appears to have quickly diminishing returns. It seems like it is possible to get queries down to 3-4 seconds per tag when searching for both measurements and notifications. Still, this is too slow for a practical application that would be used for data analysis.

The solution is most likely creating a proper, dedicated API with enough resources to handle the load from external applications. If that is not possible, it might be that the only solution is to somehow expose the underlying databases, and use data from them directly.

EqHub does not have performance issues. The Sharecat API is more than fast enough to be used for data analysis, and the server could fairly easily be rewritten to not require loading equipment on startup. The API could be made even faster if it implemented bulk requests, but that is a limitation that can be worked around. More importantly, the primary issue here is the loose and inconsistent information model. While the “class” attribute seems consistent, this is usually not sufficient for mapping to the APOS model, and so complex filters must be created.

The solution to these issues might be enforcing a detailed information model such as the APOS model. The essential requirement for such a model is that it is extensive enough to con-

tain all equipment, detailed enough that it produces consistent results, and compliant enough with other standards that mapping from the existing schemas used by equipment suppliers is not too complicated. APOS aims to satisfy these goals, but this project reinforces these requirements. Without a strict information model, different suppliers will inevitably end up with different interpretations of the model, and a practical product using the data, such as an OPC-UA server, will have to use complex filtering and exceptions to cover all possible interpretations. This is discussed in more detail in section 9.3.

While it is quite possible to design a replacement for the current setup at Aker BP, replacing something like this is expensive and difficult. There are good reasons why Aker BP have opted to not keep the existing systems, rather than redesigning everything from scratch. An AAS based system, like the one suggested in section 9.4, would be easier to work with and in the end be a cleaner information model, but the importance of a good information model is usually secondary to a system that is usable.

10.2 Discussion

In chapter 1.2, three questions were stated:

1) What requirements should be posed to APOS to make it possible to develop practical OPC-UA servers using the APOS information model?

2) Is the model developed in the specialization project sufficient to model the data found in Aker BPs systems?

3) What requirements should be posed to the contents and technical implementations of information management systems in order to make them compatible with the type of OPC-UA server described in this project?

By exploring the systems provided by Aker BP, and seeing what challenges these pose. These questions can be answered. The requirements for APOS are established in sections 9.3 and 9.5. APOS must be detailed and thorough enough that it can be applied to a system like EqHub, and enforced on all equipment. Ideally, it should provide a versioned, machine-readable document containing the full information model, where each legal field is specified.

This also raises another serious issue. Using the APOS model effectively requires a separate model for safety instrumented systems. In many cases, this prevents a major challenge for the adoption of APOS. If the APOS information model is to be used in this way, integrated with existing systems, it should either be modified to cover all equipment, or be designed in such a way that it can be easily expanded.

Another issue is also discussed in section 9.5, in that APOS may in some cases be too inflexible, and that the model run into issues with equipment that does not consistently belong to a single L2 category. In terms of question 2), this is potentially also a problem with the chosen model. It may be necessary to reconsider the base information model in a way that can handle this, potentially using a different base model than ISA-95. Either way, this potential for conflict should be addressed by the APOS project.

Finally, the source systems have for the most part been difficult to work with, which has made the creation of a common OPC-UA server a considerable challenge. As discussed above,

both the contents of the systems and the existing interfaces have created challenges for this project.

The requirements for technical systems in order to create a server like this one is simple enough: they must have fast, flexible and well documented APIs, which for now is only fully satisfied by EqHub.

Finally, for the information modeling itself. This is in many cases difficult to address. The basic requirements are simple enough: Systems must use a consistent, organized, documented data model. As seen in SAP, this means that failure notifications must not be allowed to be loosely classified. The "Other" classification is problematic and should be avoided. In order to ensure this, it is important that the information model is complex enough that valid failure classifiers can always be selected. This model could be provided by APOS, or by some other standard. This lack of specificity may also be remedied with requirements placed on the engineers that create these notifications to begin with.

The issue of relaxed requirements on information entry is also seen in EqHub. Applying a strict model like APOS to inconsistent data is difficult, and often produces results of limited quality. In EqHub especially, information entry could be subject to stricter requirements and information modeling.

10.3 Future Work

Based on the conclusions made in this project, a few general recommendations to APOS and the industry in general can be made. In addition, section 9.6 explains steps that can be taken to improve the server implementation in such a way that it can be used as part of the unified OPC-UA platform in development at Aker BP.

General Recommendations

Expand APOS to cover non-SIS equipment. As discussed in the previous section, APOS is less likely to be chosen as a standard information model if cannot be applied to non safety-critical equipment. The idea and core concepts of APOS are good, and in principle apply to all forms of equipment. In fact, the concepts of failure modes, detections and causes may be underutilized outside of safety analysis, despite being potentially useful for all equipment.

Regardless of whether APOS is expanded now, it should be designed to be expandable in general. It is inevitable that assumptions about technical solutions will change in the future, and in order to remain relevant in the future, the APOS project should define procedures for expanding the information model to cover new types of equipment.

The industry *needs* a standard model for equipment. Ideally such a model should be hosted and managed centrally, in a system like EqHub, or a larger collaboration. EqHub might be too small an initiative, being entirely Norwegian, but a larger initiative might be handled by something like the IOGP (International Association for Oil and Gas Producers). The advantage of APOS in such a setting is that it is *not* a flexible, general standard, and instead uses recommendations and guidelines from other industrial standards to create a strict, specific information model.

Ensure that existing systems are externally accessible. This is already in progress at Aker BP, and it is becoming clear that access to data for analysis, contextualization, and visualization is something industry is aware of. Whether the solution is an OPC-UA server like the one developed in this project, or extracting the data to a system like Cognite's CDF and contextualizing it later, providing automated access to live data is necessary.

If the underlying systems use non-standard but fast APIs, then a solution like an OPC-UA interface could be useful. This is effectively how the LCI database could be, if it was faster. The database requires specialized knowledge to access and understand, but an external OPC-UA interface could make it more easily accessible. As such, making the source systems accessible only really requires a fast and accessible API, not necessarily something as independent as OPC-UA.

Enforce data consistency. A solid information model is worthless if it is not obeyed. If the easiest solution for operators is not obeying the information model, then the result will inevitably be data inconsistency. The reasons for this can be many, the user interface may be difficult, or perhaps the information model is incomplete. Either way, automation rapidly loses value when faced with inconsistent data.

Further Research

Finishing the server. This will mean following the points outlined in section 9.6, to fill in the server, and integrate it with remaining Aker BP systems. This will require changes to the source systems, as discussed in chapter 9 in general.

The server implementation is licensed under GPL 2, and as such it may be freely used and expanded upon, provided the license is distributed along with the software, and that the source code is made available to anyone with a copy of the product. It is based on the official SDK, which should be relatively state-of-the-art, but it requires experience with both OPC-UA and programming in C#. Integrating it with software written with a different system would likely be difficult. In that case, it would most likely be better to re-implement the server with the methods described in this thesis.

Repeating the experiments. Using the experience from this project, the process could be repeated at another company, perhaps with a slightly different focus. Gathering more information about the problems facing industry today could provide further insight into how the APOS model should be developed in the future.

Appendix A

The Code

Unlike the specialization project, the code written for this project is not easilly runnable. It relies on access to the various highly specific internal systems in Aker BP. That said, this section will provide both technical documentation as well as instructions for reproducing the results, given access to systems like the ones in use at Aker BP.

A.1 Technical Documentation

The code is written in C#, and compiles for .NET 5.0. There are three projects:

- SourceSystems, which handles connecting to the three source systems and transforming the raw responses into something useable.
- Server, which contains the OPC-UA server itself.
- FRReader, which is the client implementation discussed in section 8.2.

Below is a general description of all provided files, there is also some inline documentation where further explanation of some methods are needed.

SourceSystems

The SourceSystems code is divided into four parts: Auth contains the code required for authenticating against Azure AD. EqHub, LCI and SAP contain code for the respective source systems. There is also a file "Config.cs", which contains classes for config deserialization.

- **Auth/IADHeaderProvider**, is an interface for obtaining Azure AD headers, the idea was that the Selenium based solution was supposed to be temporary, but as no other method ever became available, it has remained the only implementation.
- **Auth/SeleniumHeaderProvider**, uses Selenium running Google Chrome to simulate a user trying to access either the EqHub frontend, or the SAP API. It follows one of several different possible paths to try to obtain the authentication headers necessary for connecting to the source systems. Using it in the future will almost certainly require maintenance, as the authentication frontend is likely to change.

- **EqHub/EqhubScrapeSource**, uses web-scraping to extract data from EqHub. This was replaced by a proper API based solution later.
- **EqHub/EqhubSource**, uses the Sharecat API to access data from EqHub.
- **EqHub/Equipment.cs**, contains the "Equipment" class, which is used as a container for equipment attributes fetched from EqHub.
- **EqHub/IEqhubSource**, is an interface for the two different EqHub source classes, to make switching between them easier.
- **LCI/LCIAttribute.cs**, **LCI/LCIRelation.cs**, **LCI/LCITag.cs** are data classes for the four different types of LCI data: Attributes, Relations and basic Tag data, respectively.
- **LCI/LCINode.cs**, is used as a container for data relevant to a specific LCI node, it keeps track of which information has been fully retrieved, and can be created without fetching any data from the source systems, which is convenient in the server, where data is filled procedurally.
- **LCI/LCIDBSource.cs** maintains a connection to the LCI database, and uses it to extract data. Requests cannot be made simultaneously, so it uses a binary semaphore for synchronization.
- **SAP/FlocMeasurement.cs**, **SAP/FlocNotification.cs** contain the data classes for measurements and notifications respectively. Both simple data transfer classes used for deserialization of the response from the API, and slightly more advanced classes which has parsed and filtered the desired data further.
- **SAP/FunctionalLocation.cs**, contains the data transfer classes for SAP functional location. This ended up not being used, but the code to extract it exists and is fully functional.
- **SAP/SapSource.cs** is the primary class for reading from SAP. It uses an implementation of IADHeaderProvider to get the necessary authentication headers, and makes requests to the Fiori server. Requests here can be made in parallel, which can help a little with performance.
- **SAP/SAPTagData.cs** is a container class for SAP data, similar to LCINode.cs, similarly it keeps track of fetched data, to avoid fetching it twice.
- **Config.cs** contains config classes, these are filled from the various YAML files in the config folder.

The SourceSystems project does not depend on OPC-UA, and could be used for other implementations. It only depends on YamlDotNet for yaml deserialization, Selenium for authentication, and HtmlAgilityPack for the scraping-based EqHub implementation.

Server

The Server is a fairly standard OPC-UA server implementation. It does not really do anything very unusual for implementations using the official SDK.

- **Events/MeasurementEvent.cs** and **Events/NotificationEvent.cs** are event-state implementations for the two different types of SAP events. This is a fairly simple way to implement events in the OPC-UA SDK, each event contains a list of fields that they expose through overriden functions to the SDK, which handles the filtering and selection.
- **APOSServer.cs** extends the OPC-UA StandardServer class, and is the foundation for the server itself. It would make it possible to override base functionality, but in this case all it is used for is to add the two node managers.
- **APOSUtils.cs** is a utility class, for now it only contains the "DistinctBy" utility method, which is useful in a few places.
- **ExternalNodeManager.cs** is a node manager that handles the nodes fetched from SourceSystems. This means it is responsible for most of the complex functionality. It is implemented from scratch, as required by the INodeManager interface from the SDK.
- **InternalNodeManager.cs** is a much simpler node manager that extends the CustomNodeManager2 class from the SDK. This class does most of the work for in-memory nodes, meaning that it handles the APOS and ISA95 hierarchies.
- **ServerController.cs** is a utility class that contains the server itself, and manages any resources that needs disposing. It is convenient to have a class like this, so that extra server logic can be placed somewhere it does not interfere with the SDK.

The Server project obviously depends on the OPC-UA SDK, as well as the SourceSystems project.

FRReader

FRReader is a simple OPC-UA client implementation, with a few very specific methods used for the task of reading all tags belonging to an equipment group, then fetching all events from those.

The various OPC-UA methods are contained in the **FRReader.cs** class, while the overall logic is in **Program.cs**.

Config

Finally, there are a few config files contained in the "Data" directory. Some of these are largely static, the two ".xml" config files are used for configuring the server and FRReader client. There are also a few .yml files that contain various parts of the configuration.

- **config/APOS.NodeSet2.xml** contains the parts of the APOS nodeset that are not generated from config files. That is, the base types and event types.

- **config/APOS.TestClient.Config.xml** is used for configuring the FRReader client. It should largely not require any additional work.
- **config/aposstructure.yml** contains the configuration of the APOS hierarchy, as discussed briefly in section 6.2.
- **config/config.yml** contains the main configuration elements discussed in chapter 5.
- **config/knownteknos.yml** contains a list of the known TEK numbers, used for populating the hierarchy on startup. The provided version is empty, but it is written automatically by the server as new types are discovered.
- **config/Server.APOS.Config.xml** contains the server configuration, including which port it runs on.

In addition, there are two more folders. "cookies" contains a file for each site the AD Header provider is asked to obtain cookies for, storing them for future use.

The "eqhub" folder contains a file for each TEK number extracted by the scrape-based EqHub source, for efficiency. It is not used by the API-based source, since it is much faster.

A.2 Running the code

As mentioned, running the code requires access to the three source systems. It must actually be run on a machine within the same Active Directory as the LCI database in order to work properly, and the web-driver solution partly relies on this as well. Hence, in order to properly reproduce the results, it should be run on an Aker BP machine. This poses a few issues, as some part of the configuration for the Aker BP machines makes it impossible to compile .NET code, although some solution may exist.

The solution to this problem used for the purposes of this project was highly specific and will not be shared in detail here. Essentially, it meant publishing the code in a shared drive on another machine, then connecting to the Aker BP machine over SSH and running the code from there, then checking for the existence of a file to terminate the program. The code checking for the file can be found in **Server/Program.cs**.

Running the published code should not require any other dependencies at all. Compiling the code will require installing the .NET 5.0 SDK, which can be found here: `https://dotnet.microsoft.com/download`. The code also needs the file **Opc.ISA95.NodeSet2.xml** to be added to the Data/config folder, it can be found at `https://opcfoundation.org/UA/schemas/ISA-95/1.0/`.

It should be possible to compile this on windows, mac or linux. The FRReader client should run fine on any platform, but the Server code will be very difficult to make work on a non-windows machine, seeing as it relies on native windows authentication.

There are a few environment variables that should be defined for the various projects:

- "DriverPath" is the path to the google web-driver, which can be found here: `https://chromedriver.chromium.org/downloads`.

- "WebDriverUserDataDir" is the user-data directory of the web driver. Typically something like "%USERDIR%/AppData/Local/Google/Chrome/User Data/APOS".
- "APOS_ROOT_DIR" is the path to the "Data" directory discussed above.
- "EQHUB_CLIENT_ID" is the OAUTH client ID for the EqHub API.
- "EQHUB_CLIENT_SECRET" is the OAUTH client secret for the EqHub API.
- "AD_EMAIL" is the email address used with Azure AD authentication, this should be your Aker BP mail address.
- "LCI_USERNAME" is the short username for your account.
- "LCI_PASSWORD" is the password for your Aker BP account.
- "APOS_SERVER_ENDPOINT" is the URL of the server the FRReader should connect to.

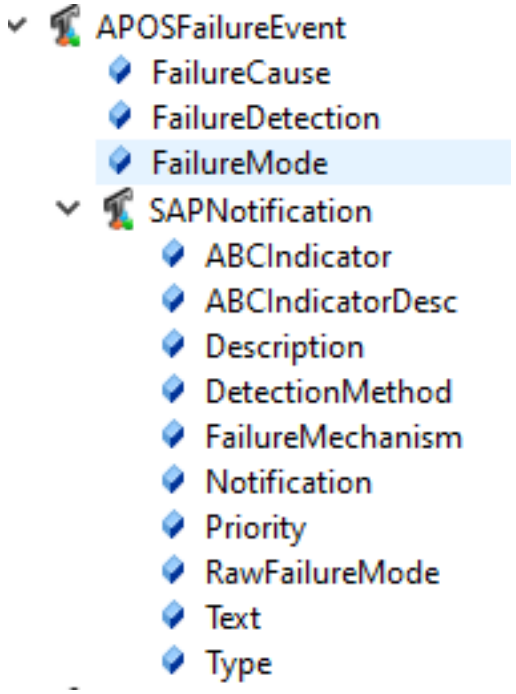
Before building, the Paket package manager must be installed by running `dotnet tool restore`, then `dotnet paket restore` can be used to download the packages.

Typically, for running the code, either use `dotnet run --project [Project]`, or `dotnet publish` and then simply run the .exe file under `[Project]/bin/Debug/net5.0/publish/[Project].exe`.

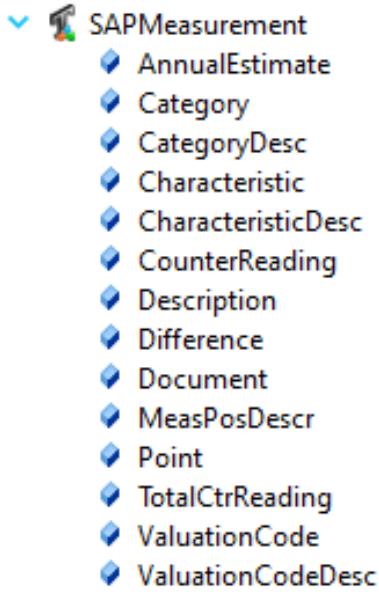
The server is, by default, hosted on port 62546, but this can be changed by modifying the file **config/Server.APOS.Config.xml**.

Appendix B

External Figures



(a) SAPNotification event type, subtype of APOSEventType



(b) SAPMeasurement event type

Figure B.1: SAP Event types in hierarchy

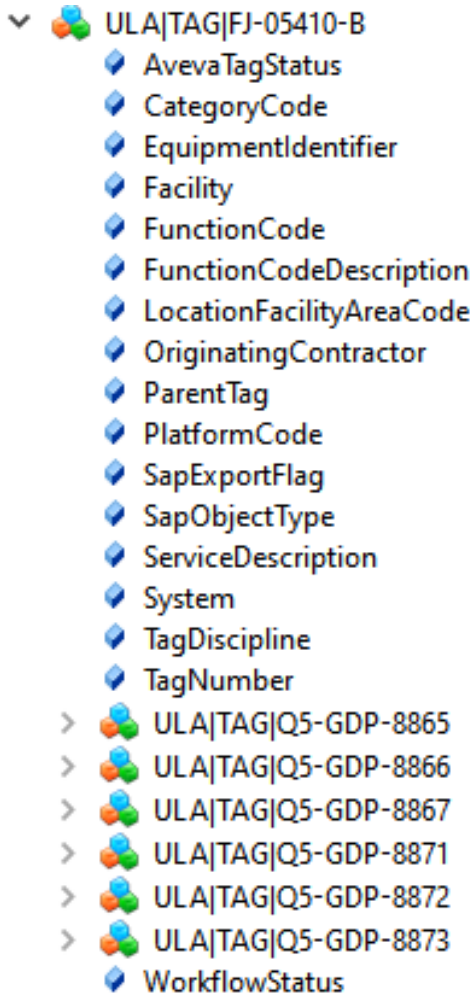
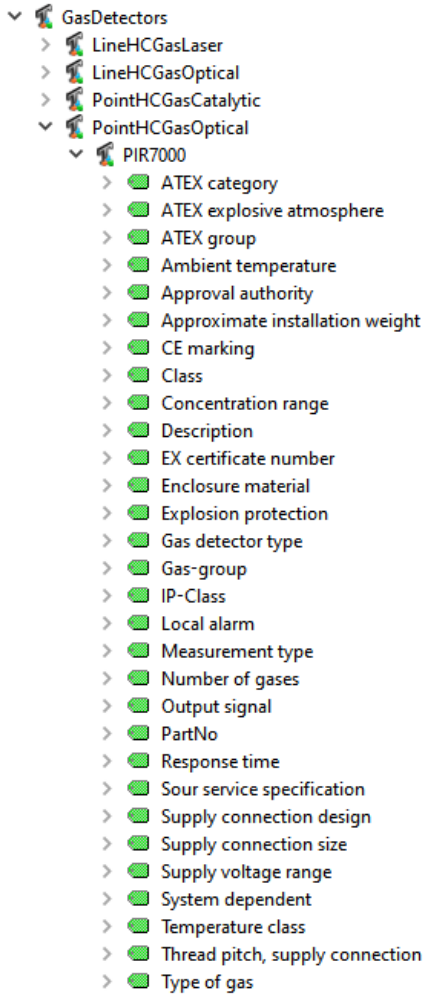


Figure B.2: LCI node in hierarchy.



(a) EqHub node in hierarchy

Reference	Target DisplayName
PhysicalAssetCl...	ULA|TAG|A1-GDP-4203
PhysicalAssetCl...	ULA|TAG|A1-GDP-4214
PhysicalAssetCl...	ULA|TAG|A8-GDP-70704
PhysicalAssetCl...	ULA|TAG|B2-GDP-4248
PhysicalAssetCl...	ULA|TAG|B2-GDP-4256
PhysicalAssetCl...	ULA|TAG|C7-GDP-4511
PhysicalAssetCl...	ULA|TAG|C8-GDP-4530
PhysicalAssetCl...	ULA|TAG|E1-GDP-4435
PhysicalAssetCl...	ULA|TAG|K10-GDP-001
PhysicalAssetCl...	ULA|TAG|M6-GDP-8803
PhysicalAssetCl...	ULA|TAG|M7-GDP-8815
PhysicalAssetCl...	ULA|TAG|O3-GDP-8778
PhysicalAssetCl...	ULA|TAG|O3-GDP-8786
PhysicalAssetCl...	ULA|TAG|Y2-GDP-8774
PhysicalAssetCl...	ULA|TAG|Q5-GDP-8867
PhysicalAssetCl...	ULA|TAG|Q6-GDP-8884
PhysicalAssetCl...	ULA|TAG|R1-GDP-8899
PhysicalAssetCl...	ULA|TAG|R1-GDP-8907

(b) References to LCI tags on EqHub Node

Figure B.3: EqHub Node

Name	Value
AnnualEstimate	-12
Category	B
CategoryDesc	Barrier Relevant
Characteristic	Check response of gas detector
CharacteristicDesc	Check response of gas detector
CounterReading	-12
Description	GTB VENT OUTL AUX ROOM
Difference	-12
Document	193123
EventId	len=16, 0x63f20200000000003e15020000000000
EventType	NodeId
NamespaceIndex	4
IdentifierType	String
Identifier	SAPMeasurement
MeasPosDescr	CHECK RESPONSE OF GAS DETECTOR
Message	"", "GTB VENT OUTL AUX ROOM, CHECK RESPONSE OF GAS DETECTOR: "
Point	136510
ReceiveTime	10:54:34.960
Severity	100
Time	08:42:45.000
TotalCtrReading	-12
ValuationCode	1.1
ValuationCodeDesc	PASS

Figure B.4: SAP MeasurementDocument received by UAExpert

Name	Value
ABCIndicator	A
ABCIndicatorDesc	High
Description	Detektor låg med feil
DetectionMethod	MADM0001-0005 - (Detection Method -Continuous condition monitoring)
EventId	len=8, 0x9cddd81700000000
▾ EventType	NodeId
NamespaceIndex	4
IdentifierType	String
Identifier	SAPNotification
▾ FailureCause	NodeId
NamespaceIndex	2
IdentifierType	String
Identifier	BaseFailureCause
▾ FailureDetection	NodeId
NamespaceIndex	2
IdentifierType	String
Identifier	Diagnosed/ImmediatelyDetected-Event
FailureMechanism	FM000005-5.2 - (External influence - Contamination)
▾ FailureMode	NodeId
NamespaceIndex	2
IdentifierType	String
Identifier	MinorInServiceProblems
Message	"", "Detektor låg med feil: Vasket og nulljustert, OK"
Notification	400088476
Priority	3
RawFailureMode	FG000001 -SER - Fire and gas detectors -Minor in-service problems
ReceiveTime	10:54:34.960
Severity	500
Text	Vasket og nulljustert, OK
Time	18:05:36.000
Type	M2

Figure B.5: SAP Notification received by UAExpert, note the NodeIds of the APOS fields, these have been mapped over.

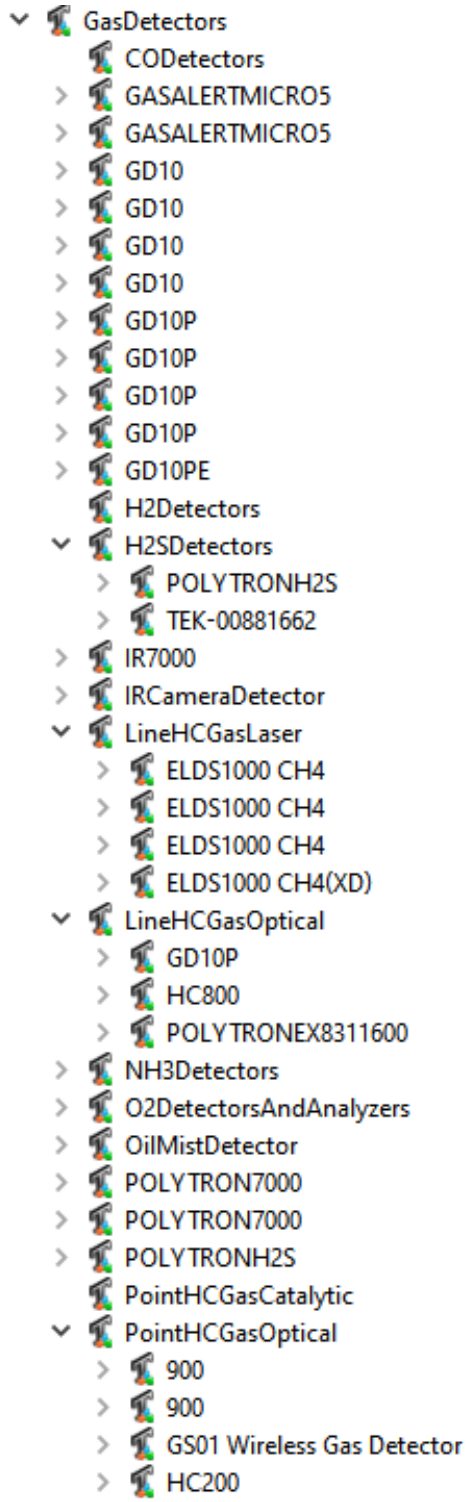


Figure B.6: Populated gas detector hierarchy.

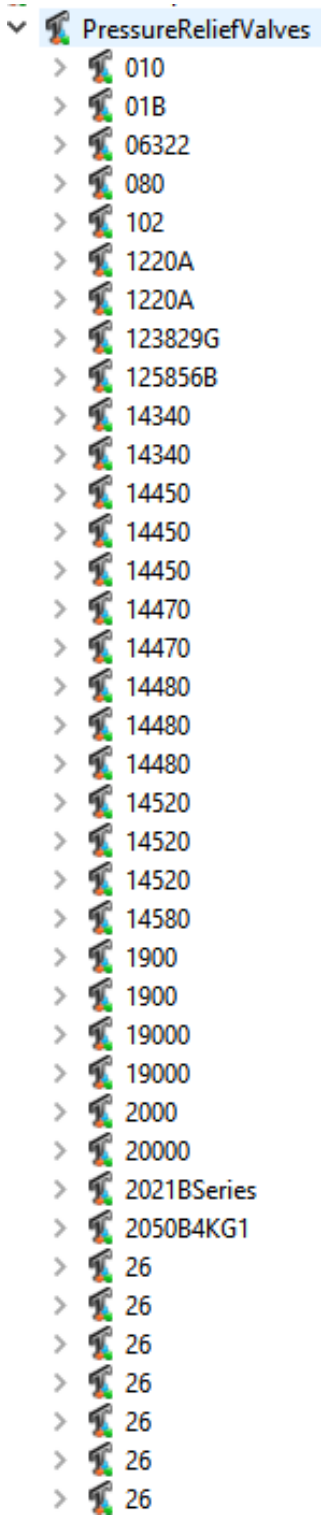


Figure B.7: Populated PSV hierarchy.

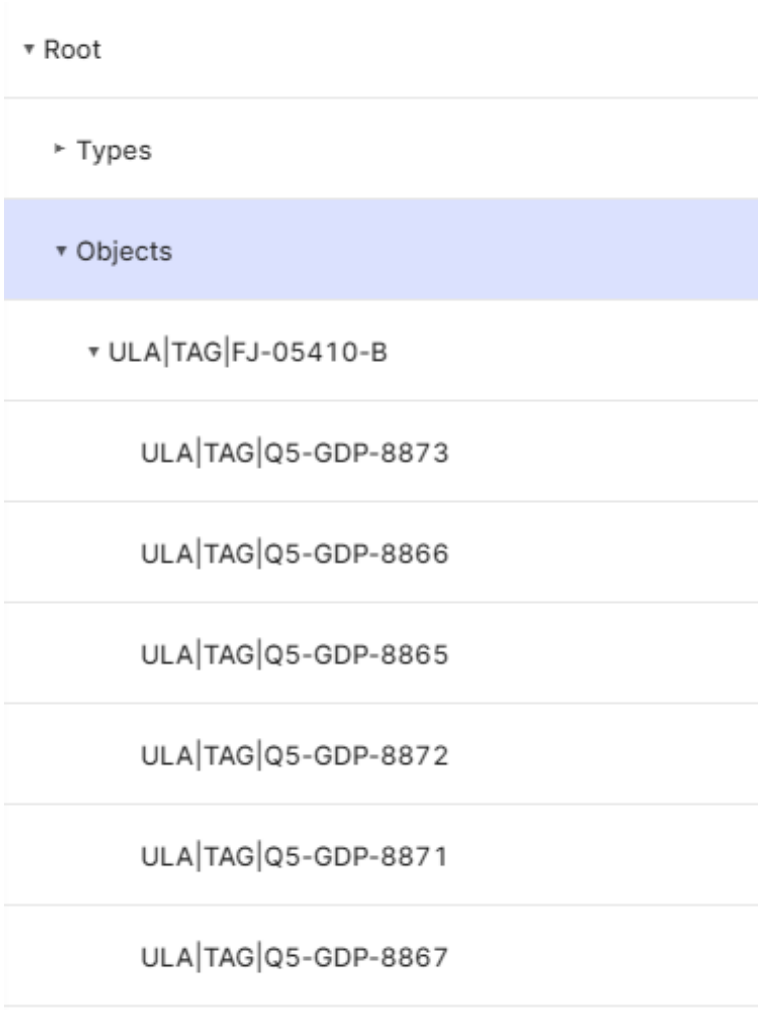


Figure B.8: LCI node in Fusion

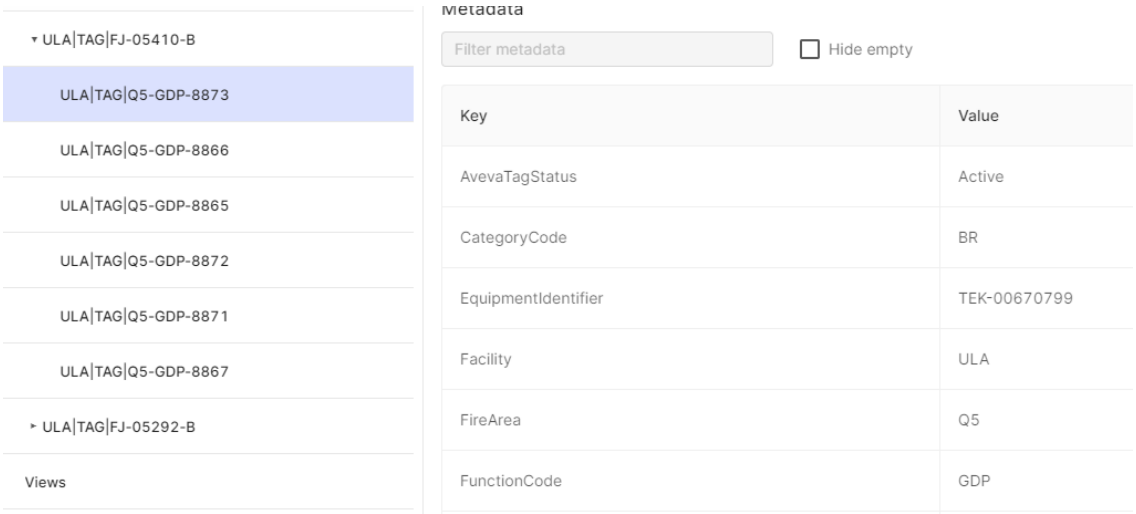


Figure B.9: LCI node metadata in Fusion

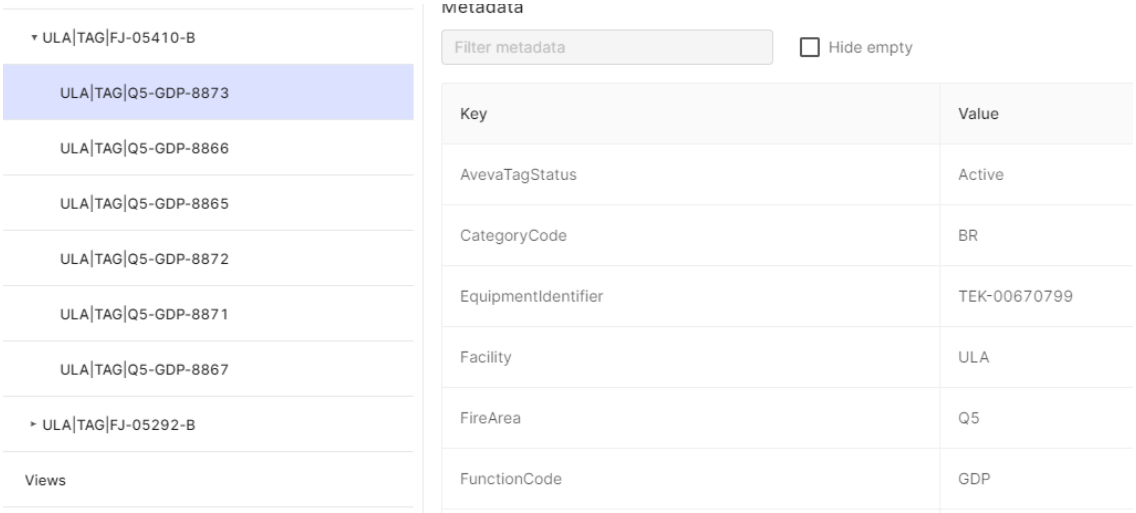


Figure B.10: EqHub node metadata in Fusion

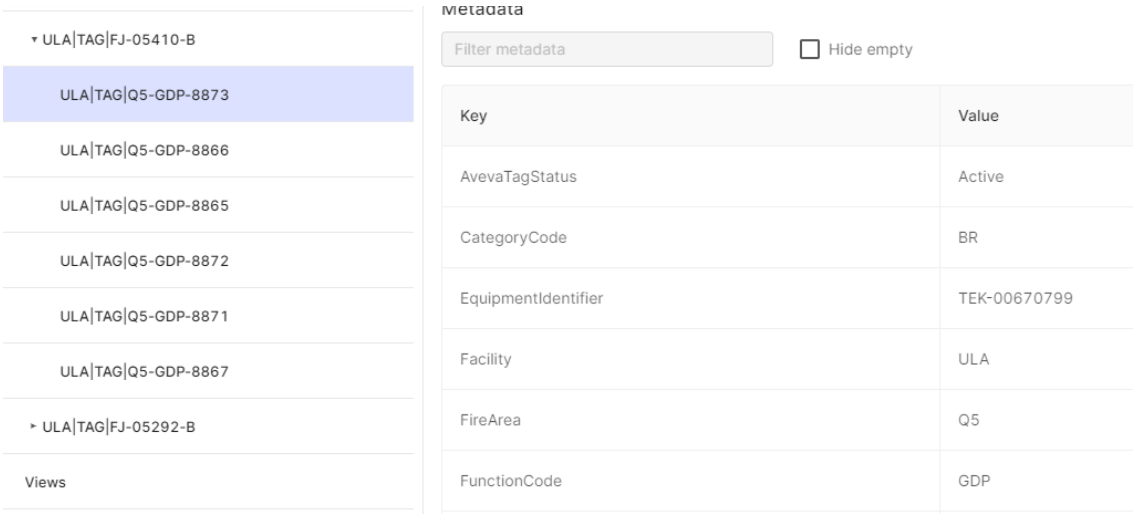


Figure B.11: Measurement event in Fusion

Appendix C

Acronyms

APOS Automatisert prosess for oppfølging av instrumenterte sikkerhetssystemer (Eng: Automated process for follow-up of safety instrumented systems)

SIS Safety Instrumented System

SIF Safety Instrumented Function

PLC Programmable Logic Controller

DU Dangerous Undetected

DD Dangerous Detected

S Safe

SDK Software Development Kit

OPC-UA Open Protocol Communications - Unified Architecture

OPC (previously) OLE for Process Control

HTTP Hyper Text Transfer Protocol

HTTPS HTTP Secure

TCP Transmission Control Protocol

LCI Life Cycle Information

SAP Systems, Applications, and Products in Data Processing

DETNOR Det Norske Oljeselskap (The Norwegian oil company)

HTML HyperText Markup Language

JS JavaScript

JSON JavaScript Object Notation

ISOC Internet Society

IETF Internet Engineering Task Force

RFC Request For Comment

W3C World Wide Web Consortium

API Application Programming Interface

GUID Globally Unique Identifier

URL Universal Resource Locator

XML Extensible Markup Language

SOAP Service Oriented Architecture Protocol

ODBC Open Database Connectivity

SQL Structured Query Language

AD Active Directory

REST Representational State Transfer

SSO Single Sign On

YAML YAML Ain't Markup Language

PSV Pressure Safety Valve

Bibliography

[1] Norsk olje og gass. URL https://www.norskoljeoggass.no/.

[2] YamlDotNet. URL https://github.com/aaubry/YamlDotNet.

[3] Soap version 1.2. apr 2007, (Accessed March 2021). URL https://www.w3.org/TR/soap12/.

[4] *IEC-61508 Functional safety of electrical-electronic-programmable electronic safety-related systems*. IEC, 2010.

[5] What is azure active directory?, may 2020, (Accessed March 2021). URL https://docs.microsoft.com/en-us/azure/active-directory/fundamentals/active-directory-whatis.

[6] Regular expression language - quick reference, mar 2021, (Accessed March 2021). URL https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference.

[7] ASP .NET, (Accessed March 2021). URL https://dotnet.microsoft.com/apps/aspnet.

[8] Aveva, (Accessed March 2021). URL https://www.aveva.com/.

[9] Cognite, (Accessed March 2021). URL https://www.cognite.com/.

[10] Cognite documentation, (Accessed March 2021). URL https://docs.cognite.com/.

[11] Sharecat TEK API 2.0, (Accessed March 2021). URL https://api.sharecat.com/v2/help/index.html.

[12] EqHub - EPIM, (Accessed March 2021). URL http://epim.no/eq-hub/.

[13] SAP Fiori, (Accessed March 2021). URL https://www.sap.com/products/fiori.html.

[14] HTML & CSS, (Accessed March 2021). URL https://www.w3.org/standards/webdesign/htmlcss.

[15] W3C JavaScript Web APIs, (Accessed March 2021). URL https://www.w3.org/standards/webdesign/script.html.

[16] ODATA, (Accessed March 2021). URL https://www.odata.org/.

[17] SAP, (Accessed March 2021). URL https://www.sap.com/norway/index.html.

[18] Sharecat, (Accessed March 2021). URL https://www3.sharecat.com/.

[19] UA-.NETStandard Samples, (Accessed March 2021). URL https://github.com/OPCFoundation/UA-.NETStandard-Samples.

[20] UA-.NETStandard, (Accessed March 2021). URL https://github.com/OPCFoundation/UA-.NETStandard.

[21] W3C URI Specification, (Accessed March 2021). URL https://www.w3.org/Addressing/URL/uri-spec.html.

[22] W3C Standards, (Accessed March 2021). URL https://www.w3.org/standards/.

[23] YAML ain't markup language, (Accessed March 2021). URL https://yaml.org/.

[24] D. D. Chamberlin. Early history of sql. *IEEE Annals of the History of Computing*, 34(4): 78–82, 2012. doi: 10.1109/MAHC.2012.61.

[25] International Electrotechnical Commission. *IEC-61511 Functional safety: Safety instrumented systems for the process industry*. International Electrotechnical Commission, 2018.

[26] International Electrotechnical Commission. IEC-60050 international electrotechnical vocabulary, (Accessed April 2021). URL https://www.electropedia.org/iev/iev.nsf/index?openform&part=192.

[27] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol - http/1.1. Jun 1999 (Accessed March 2021). URL https://tools.ietf.org/html/rfc2616.

[28] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures (PHD). 2000.

[29] International Organization for Standardization. *ISO-14224: Petroleum, petrochemical and natural gas industries: collection and exchange of reliability and maintenance data for equipment*. International Organization for Standardization, 2006.

[30] Dick Hardt. The oauth 2.0 authorization framework. Apr 2012 (Accessed March 2021). URL https://tools.ietf.org/html/rfc6749.

[31] Courtney Hunt. Understanding the world wide web: A brief primer, Mar 2014 (Accessed March 2021). URL https://www.socialmediatoday.com/content/understanding-world-wide-web-brief-primer.

[32] Leach, P. and Mealing, M. and Salz, R. A universally unique identifier (uuid) urn namespace. Jul 2005 (Accessed March 2021). URL https://tools.ietf.org/html/rfc4122.

[33] Microsoft. Microsoft Open Database Connectivity (ODBC), 2016 (Accessed March 2021). URL `https://cdn.simba.com/wp-content/uploads/2016/03/ODBC_specification.pdf`.

[34] Einar Marstrander Omang. APOS OPC-UA. Dec 2020.

[35] OPC Foundation. Opc ua online reference, Feb 2021 (Accessed March 2021). URL `https://reference.opcfoundation.org/v104/`.

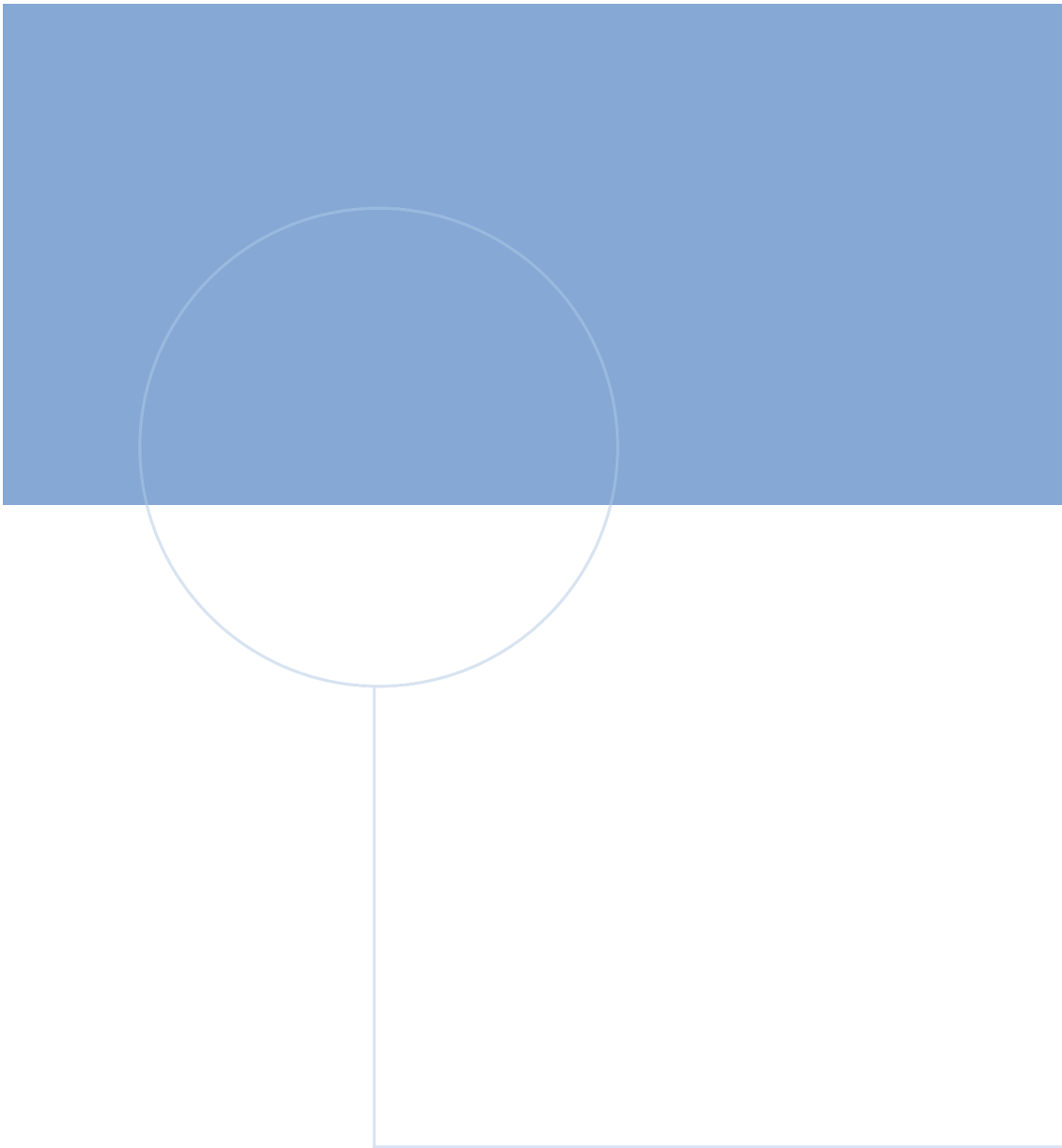
[36] Marvin Rausand. *Reliability of safety-critical systems: theory and applications*. Wiley, 2014.

[37] SINTEF. Standardised failure reporting and classification of SIS failures in the petroleum industry - draft version 3.0. Dec 2019.

[38] SINTEF. H5 - APOS object model specification - preliminiary draft. Apr 2020.

[39] SINTEF. Automatisert prosess for oppfølging av instrumenterte sikkerhetssystemer, (Accessed May 2021). URL `https://www.sintef.no/prosjekter/automatisert-prosess-for-oppfolging-av-instrumenterte-sikkerhetssystemer/`.

[40] Constantin Wagner, Julian Grothoff, U. Epple, Rainer Drath, S. Malakuti, Sten Grüner, M. Hoffmeister, and P. Zimmermann. The role of the industry 4.0 asset administration shell and the digital twin during the life cycle of a plant. *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2017.



NTNU

Kunnskap for en bedre verden